Master's Thesis

submitted by

Martin Mosbeck

Registration Number 0926555

Subcircuit Pattern Matching in Digital Designs

In partial fulfillment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Vienna, Austria, April 15, 2020

Study code:	066 504
Field of study:	Embedded Systems

Supervised by:	Univ. Ass. Dipl.-Ing. Dr.techn. Christian Krieg
	Univ. Prof. Dipl.-Ing. Dr.techn. Axel Jantsch

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Copyright (C) 2020 Martin Mosbeck

If you find this work useful, please cite it using the following BibTeX entry:

```
@Thesis{Mosbeck2020,
 type        = {Master's Thesis},
 author      = {Martin Mosbeck},
 title       = {Subcircuit Pattern Matching in Digital Designs},
 school      = {Vienna University of Technology (TU Wien)},
 year        = {2020},
 address     = {Gusshausstrasse 27--29 / 384, 1040 Wien},
 month       = {04},
}
```

Contact us:

martin.mosbeck@gmx.at

christian@drkrieg.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Abstract

Design understanding is an essential part of reverse engineering and verification of digital designs. Extracting and subsequently analyzing subcircuits of synthesized designs offers an alternative approach to simulation and analyzing hardware descriptions. In this thesis we develop a search algorithm to extract functional blocks like counters and state machines from synthesized *Verilog* designs via structural pattern matching. Extracting functional blocks supports reverse- and verification engineers in checking a given design against its specification, identify errors, perform security evaluations, and deepen overall understanding of a design.

By analyzing the related work we find three approaches for finding subcircuits in a design: (1) matching based on functional equivalence, (2) matching based in structural equivalence, and (3) mixed approaches. We discuss each of the approaches and analyze their deficiencies. Functional approaches struggle with capturing the full and characteristic functionality of subcircuits to compare them to reference functionality descriptions. Capturing and comparing the full and characteristic functionality may require exhaustive simulation in the worst case. Structural methods struggle with capturing and identifying structural variability of components with similar functionality. Mixed approaches aim at combing the strength and weaknesses of both approaches, resulting in one algorithm per functional class. We extend the state of the art by introducing one single algorithm that efficiently matches a pattern to capture all representatives of a functional class. These patterns use serial and parallel quantification of occurrence of subpatterns to model structural variability.

We use the pattern graph specification language (PGSL) to describe functional classes for which we search representatives in an abstraction of a digital design called *design graph*. Design abstraction significantly reduces the complexity of the search problem. We demonstrate pattern modeling on the practical examples state machines, counters and elements of encoders/decoders. The proposed search algorithm searches for pattern matches by subgraph isomorphism. It parses PGSL, effectively filters the search space based on the chosen pattern and uses a search-and-combine approach to find pattern matches. We use a custom constraint satisfaction problem (CSP) solver to combine matches of subpatterns until full matches are found. We implement our search algorithm as plugin to the open-source synthesis suite *Yosys*.

To demonstrate the effectiveness and efficiency of our chosen search approach we conduct a large-scale experiment on 74 open-source designs of increasing complexity and five patterns covering all important features of PGSL. Experimental results show satisfactory search times in the range of seconds to minutes and search results can be used for further design analysis. We demonstrate the usefullness of results for example patterns and designs.

Furthermore, we analyze the performance of our algorithm. Subgraph isomorphism is an NP-complete problem which has exponential worst case behavior. Performance estimation is complicated by the fact that we do not search matches for one uniform graph, but rather pattern graphs that are nested and contain quantified subpatterns. This leads to interesting side effects like the search for quantified subpatterns dominating the search. In these cases the subpattern search results in a large amount of matches in the design that all have to be checked during the combination step of the search algorithm.

Summing up, we show that we are able to do what has not been possible before: We are able to model abstract, regular-expression-like patterns for subcircuits, search them in an abstract graph of real-world designs, and extract candidate subcircuits that match such patterns.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kurzfassung

Das Verstehen von digitalen Designs stellt einen wesentlichen Bestandteil von Reverse Engineering und der Verifikation von digitaler Designs dar. Ein alternativer Ansatz zu Simulation und Analyse von Hardwarebeschreibungen bietet das Extrahieren und anschließendes Analysieren von Teilschaltungen synthetisierter Designs. Der Fokus der vorliegenden Arbeit liegt auf der Entwicklung eines Suchalgorithmus um funktionale Blöcke wie Zähler und Zustandsautomaten aus synthetisierten Verilog-Designs mittels strukturellem Mustervergleich zu extrahieren. Die Extraktion von Funktionsblöcken unterstützt Reverse- und Verifikationsingenieure ein gegebenes Design mit dessen Spezifikation zu vergleichen, bei der Identifizierung von Fehlern, bei der Durchführung von Sicherheitsüberprüfungen, sowie der Vertiefung des Gesamtverständnisses eines Designs.

In einer umfassenden Literaturstudie konnten drei Ansätze zur Identifizierung von Teilschaltungen in einem Design identifiziert werden: (1) Matching basierend auf funktionaler Äquivalenz, (2) Matching basierend auf struktureller Äquivalenz und (3) gemischte Ansätze. Die vorliegenden Ansätze werden im Zuge dieser Arbeit im Detail analysiert und im Hinblick auf ihre Mängel diskutiert. Funktionale Ansätze sind problematisch im Hinblick auf das Erfassen der vollständigen und charakteristischen Funktionalität von Teilschaltungen und deren Vergleich mit Referenzfunktionsbeschreibungen. Um die vollständige und charakteristische Funktionalität zu erfassen und zu vergleichen, kann im schlimmsten Fall umfangreiche Simulation notwendig sein. Strukturelle Methoden wiederum zeigen Schwierigkeiten bei der Erfassung und Identifizierung von struktureller Variabilität bei Komponenten mit ähnlicher Funktionalität. Gemischte Ansätze, die versuchen die Stärken der beiden zuvor genannten Ansätze zu kombinieren, führen allerdings zu jeweils einem Algorithmus pro funktionaler Klasse. Das Ziel dieser Arbeit ist die Entwicklung eines einzigen, effizienten Algorithmus zum Mustervergleich, wobei ein Muster jeweils alle Vertreter einer funktionalen Klasse erfasst. Diese Muster verwenden serielle und parallele Quantifizierung um Strukturvariabilitäten als quantifiziertes Auftreten von Submustern zu modellieren.

Um Funktionsklassen zu beschreiben, die wir in einer Abstraktion von digitalen Designs namens *design graph* suchen, verwenden wir die pattern graph specification language (PGSL). Die Designabstraktion bietet den Vorteil einer erheblichen Reduktion der Komplexität des Suchproblems. Die Funktionsweise der Mustermodellierung wird an praktischen Beispielen wie Zustandsmaschinen, Zählern und Elementen von Encodern/Decodern demonstriert. Der in dieser Arbeit entwickelte Algorithmus sucht nach Musterübereinstimmungen mittels Subgraph-Isomorphismus. Der Suchalgorithmus analysiert das PGSL Muster, filtert den Suchraum basierend auf dem ausgewählten Muster und verwendet einen Such- und Kombinationsansatz für die Identifizierung von Musterübereinstimmungen. Wir verwenden einen benutzerdefinierten constraint satisfaction problem (CSP) Löser um Übereinstimmungen von Submuster zu kombinieren bis vollständige Übereinstimmungen für das gesamte Muster gefunden werden. Der Suchalgorithmus wurde als Plugin

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

für die Open-Source Synthesissuite *Yosys* implementiert.

Um die Effizienz und Funktionsfähigkeit des entwickelten Suchalgorithmus zu demonstrieren wird ein umfassendes Experiment durchgeführt. Dies umfasst 74 Open-Source-Designs mit zunehmender Komplexität und 5 Muster welche die wichtigsten Funktionen von PGSL abdecken. Die Ergebnisse der zuvor genannten Untersuchung zeigen zufriedenstellende Suchzeiten im Bereich von Sekunden bis Minuten. Des Weiteren können die Suchergebnisse für weitere Designanalysen verwendet werden. Die Nutzbarkeit der Ergebnisse für Beispielmuster und Designs wird gezeigt.

Darüber hinaus befasst sich die vorliegende Arbeit mit der Performance des entwickelten Algorithmus. Bei Subgraph-Isomorphismus handelt es sich um ein NP-vollständiges Problem mit exponentiellem Worst-Case Verhalten. Die Abschätzung der Performance des Suchalgorithmus wird durch die Tatsache erschwert, dass keine Übereinstimmung für einen einheitlichen Graphen gesucht wird, sondern für Mustergraphen die verschachtelt sind und quantifizierte Submuster enthalten. Dies führt zu interessanten Nebeneffekten wie z.B., dass die Suche nach quantifizierten Submuster die Suche dominiert. In diesen Fällen führt die Suche nach Submuster zu einer großen Anzahl von Übereinstimmungen im Design, die alle während des Kombinationsschrittes des Suchalgorithmus überprüft werden müssen.

Zusammenfassend zeigen wir, dass wir in der Lage sind, das zu tun, was bisher nicht möglich war: Wir sind in der Lage, abstrakte, an regular expressions angelehnte Muster für Teilschaltungen zu modellieren, sie in einer Graphabstraktion von Designs aus der realen Welt zu suchen und die Suchergebnisse als Teilschaltungen zu extrahieren.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acknowledgements

At last the journey is at an end. A journey with ups and downs, with times of despair, but also times of content, when things were finally working out as I intended them to. This thesis not only gives me the opportunity to present the interesting work I have done, but also to thank those that gave me guidance, support, those who diverted me to the paths that lead me to reaching my destination.

A special thanks goes out to:

My supervisors Axel Jantsch and especially Christian Krieg for their support, input, feedback, guidance and discussions that helped me to shape this thesis.

My mother Helga, my father Manfred, and my sister Margit both for financial and moral support. I am glad I have a family that will always stay by my side, support me, listen to my problems, give advice or just plainly listen. Thank you for everything!

My friends Davor Frkat, Christoph Peinsipp and Andreas Potucek who accompanied me in my studies. You are true friends, the kind of friends that everyone should have.

My colleagues and friends at the Fachschaft Elektrotechnik for the many nights of conversations and hours of collegiality.

All my friends and my sister who invested hours to proofread my thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Contents

1 Introduction **1**
1.1 Problem statement and proposed solution 1
1.2 Research questions and thesis structure 6
1.3 Contributions 7

2 Related work **9**
2.1 Low-level reverse engineering 10
2.2 Finding high-level components 11
2.3 Summary 16

3 Background **17**
3.1 Design graph abstraction 17
3.2 Constraint satisfaction problem 19
3.3 Subgraph isomorphism 22
3.4 Regular expressions 24
3.5 Regular graph expressions 26

4 Search pattern modeling **29**
4.1 Structural elements of PGSL 30
4.2 Creating PGSL patterns 33

5 Search methodology and implementation **37**
5.1 Parsing 38
5.2 Creating the filtered design graph 40
5.3 Hierarchical creation and combination of candidates 40
5.4 Result post-processing and visualization 54

6 Experiments and results **55**
6.1 Experimental setup 55
6.2 Experimental results 58
6.2.1 State machines 62
6.2.2 Counters 65
6.2.3 Encode/decode elements 68

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

7 Discussion 71

8 Conclusion 75

Bibliography 81

Appendices 85

A State machine Verilog code packet assembly in USB core 85

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Tables

1.1 Comparison of the terminology concerning PGSL between this thesis and the PhD thesis of Christian Krieg.[1] The terminology used in this thesis reflects the implementation of the developed search algorithm. . . . 4

3.1 Selected metacharacters used with IEEE POSIX ERE. . . . 25

4.1 Available quantifiers in PGSL and their meaning. Quantifiers specify the number of occurrences of a quantified block to match. . . . 33

5.1 Average decrease of the search space parameters number of nodes and number of connections after filtering to types contained in a given PGSL pattern. The decrease percentages are averaged for the searches in the 74 designs and separated for each of the five patterns we use in the experimental chapter (see Chapter 6). . . . 40

6.1 Statistics collected by the search plugin during a search operation. . . . 56

6.2 Overview of the five patterns used to search in the *Verilog* designs obtained from *OpenCores*. . . . 57

6.3 Abbreviations and full names used for the *OpenCores* categories. . . . 59

6.4 Overview of the experimental results. For each pattern, *S* denotes the number of search results and *P* the number of modules after post-processing. *t(S)* and *t(P)* are the associated processing times. The last column *t* sums up the total processing time for each design. Results marked in bold are discussed in depth in the following sections. . . . 60

6.4 - continued from previous page . . . 61

6.5 Comparing the search space for the designs *wb_conmax* and *usb*. To save space we abbreviate the design *wb_conmax* as *wb*. Highlighted are the number of cells, for cell types, that are matched in the dominant subpattern (($mux | $logic_not) |* || $eq |+ 65

6.6 Comparing the search space and performance for the designs *reed_solomon_decoder* and *m32632*. . . . 68

6.7 Comparing the search space and performance for the designs *spimaster*, *m32632*, and *jt51*. Rising search time correlates with rising number of $eq and \CO cells and a higher number of connections. . . . 70

7.1 Duration statistics for the pattern-search and post-processing operations (experiment with 74 designs, 5 patterns, description in Chapter 6). The percentages are rounded. . . . 71

[1] C. Krieg. “Pattern-Based Hardware Trojan Characterization for Design Security Assessment”. PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

7.2 Overall impact of post-processing. For every pattern we show the number of designs that have search results for this pattern, and the number of those searches that were impacted by post processing. Post-processing had an impact if the number of results after post processing was smaller due to merging of search results. 72
7.3 Average decrease of the search space parameters number of nodes and number of connections after filtering to types contained in a given PGSL pattern. The decrease percentages are averaged for the searches in the 74 designs and separated for each of the five patterns we use in the experimental chapter (see Chapter 6). 74
7.4 Impact of high connectivity for quantified subpattern searches. We compare the search space and performance for searches with pattern *Counter 2* for the designs *reed_solomon_decoder* and *m32632*. 74

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures

1.1 The RTL intermediate language (RTLIL) in graph representation of the example first in first out (FIFO). At RTLIL the module is synthesized and contains cells with ports and nets connecting ports. 3
1.2 The design graph abstraction of the example FIFO. The result is a directed graph consisting of nodes with types and connections between these nodes. 4
1.3 Graph view of the PGSL pattern for counters. 5
1.4 Overview of the architecture proposed in this thesis. A *Verilog* hardware description language (HDL) design is synthesized using *Yosys* and abstracted to a design graph. Search patterns modeled in PGSL are run through a parser and used to first filter the design to "types of interest" i.e. the types the pattern contains. This pattern and the filtered graph are used together in a pattern searcher, that constructs search results, that match the pattern in the design graph. A similar architecture overview can also be found in the PhD thesis of Krieg[2] as that work also discusses the algorithm that was developed in this thesis. 6

2.1 Different approaches to find high-level components in designs and the related work that we will discuss. Functional matching finds subcircuits whose function matches a component in a library. Structural matching checks for subgraph isomorphism. Hybrid matching combines both methods in multi-step approaches. 12

3.1 Visualization of an example model transformation from register transfer level (RTL) netlist to design graph abstraction. With this abstraction input and output ports are omitted and individual wires are merged to directed edges. 18
3.2 Visualization of an example map-coloring problem. Figure 3.2a is a fictional map with seven territories labelled A to F. In map coloring we have to color each territory with a color from a given set, such that no adjacent territories have the same color. Figure 3.2b is a visualization of the associated constraint graph. Each vertex represents one of the territories and each edge represents a coloring constraint between two adjacent territories. 19
3.3 Backtracking search tree for our example map-coloring problem. The numbers indicate the traversal of the tree. Assignments marked with X violate a constraint and cause a backtrack. 20

4.1 The grammar of the PGSL in backus–Naur form (BNF) 30

[2] C. Krieg. “Pattern-Based Hardware Trojan Characterization for Design Security Assessment”. PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

4.2 Illustrating the block structure of a PGSL pattern. Block (1), (2), and (4) are single nodes, block (3) is a composite node containing block (1) and (2). (0) is the whole pattern, that itself is an unquantified composite node. 31

4.3 Visualizing the use of *outer operators* in PGSL patterns and there equivalent graphs. 32

4.4 *Yosys* RTLIL graph of a simple arbiter state machine with three states (encoded as "1", "10", "100"). The cells we choose as part of the structural kernel are marked. We use this structural kernel as first version for a PGSL pattern for state machines. 34

5.1 Graphical representation of the pattern shown in Equation (5.1). 39

5.2 Illustrating the result of parsing for an example pattern. The pattern is parsed from left to right, the *pn_id* is incremented starting at 1 (*pn_id=0* is reserved for the top composite node). (1) and (2) in the pattern are parsed as single nodes and combined to a composite node (3), that contains the two nodes and their connection information. (4) is parsed to a single node. The pattern as a whole (0) forms a composite node as top-level. The pattern graph structure stores the top *pn_id* and the contained node types of the pattern. 39

5.3 First three graphs the pattern $add -> $mux >* -> $dff defines. The quantification >* specifies that the subpattern $mux should be matched zero or more times in serial. 41

5.4 Example pattern, which we use to illustrate the traversal involved in the search process for this pattern. The numbers are the pattern-node IDs. 41

5.5 High-level view of our search-and-combine methodology. Starting with the top composite node we resolve a composite node by stepping into it (1). Single nodes are always resolved first (2). If the composite node contains other composite nodes, they are resolved next by recursively stepping into them (3). If all nodes of a composite node have been resolved we use our CSP solver to combine the candidates of the contained nodes to a candidate for the composite node (4). If the composite node is quantified, we next resolve the quantification (5). When a composite node is fully resolved and has an non empty set of candidates, we can step out of the composite node (6). In the case of the top composite node, stepping out at (6) terminates the search operation with the created candidates being the results of the search. If at any point in the search process an empty set of candidates for a node is created, the search terminates with no results (7). A similar flow chart can also be found in the PhD thesis of Krieg[3] as that work also discusses the algorithm that was developed in this thesis. 43

5.6 Types of candidates we use in our search operations. Only the important attributes of each class are shown. *Candidate* is the abstract superclass for all candidate types. *NodeId* is an ID of a node in the design graph, *PatternNodeId* is an ID of a node in the pattern graph and *CandidateId* (as variable *c_id*) is the unique ID we assign each candidate we create. 44

5.7 Visualizing the three cases for valid parallel quantification. The parallel-quantified nodes either need a common branch, a common sink, or both. 46

5.8 Constraint graph for combining candidates of the contained nodes in the pattern [[$a -> $b -> $c]]. Each of the vertices represents one of the contained single nodes of this pattern. Vertices connected by an edge have a common constraint which is labeled on the edge. 47

[3] C. Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

5.9 Visualizing the setup for our simple CSP. From left to right: filtered design graph, candidates table, CSP-vertices table with candidates per vertex. The node IDs (n_id) in the design graph are saved in candidates with unique candidate IDs (c_ids). As a result, each CSP vertex with an unique CSP ID (csp_id) has a set of candidates (its domain). In order to make the solving process easier to understand, this fictional created design graph has been created in a way so a candidate with *c_id=x* has a contained node candidate with *n_id=x*. This is not usual for real design graphs. 48

5.10 Tree that visualizes part of the backtracking search for our example CSP and the constraints as constraint graph. Starting from an empty assignment (0), first the vertex with *csp_id*=1 is assigned its first candidate (1). Subsequently the assignment is extended (2), (3). The traversal in this search tree is depth first. Backtracking is performed if a complete and consistent assignment (check-mark marks a solution; e.g., (3)) has been found, a constraint violation occurred (X marks a constraint violation; e.g., (4), (5), (6)) or a vertex has no more possible candidates to assign. After each backtrack the next possible candidate for the to be assigned vertex is added to the assignment. 48

5.11 Classes and types involved in setting up a CSP backtracking search. Only the important attributes and methods of each class are shown. *CSPVertex* is the abstract superclass for all other CSP-vertices. The *IntersectContainer* class plays a core role in finding the common parallel candidates (intersection) for parallel-quantified nodes, if the node is constrained by more than one branch and/or sink. 51

6.1 Architecture of the *usb* core (line 62 in Table 6.4). (a) shows the overall architecture of the core, (b) shows the architecture of the protocol layer (PL) block. The numbers denote where and how many state machines we found in the modules of the architecture block. The images are taken from the documentation of the core,[4] the numbers are added. 62

6.2 Solution as subgraph of the design graph for state machine in the packet assembly module of the *usb* design. 63

6.3 Parallel state machine structure found in the *ca_prng* design. Each of the 32 parallel state machine takes three bits of the $dff register cell as state value and calculates one new bit for the register. Post-processing detects this as one state machine. 64

6.4 State machine structure with different feedback paths found in the *nextz80* design. Each of the five different feedback $mux cell chain paths is a distinct result of the search operation. Post-processing detects this as one state machine. 64

6.5 Architectural view of a simple asynchronous serial controller (line 47 in Table 6.4). The top design instantiates one FIFO for receiving data and one FIFO for transmitting data. Each of the FIFO contains two counters. Additionally, the top design contains two counters to count the position of the transmitted and received bit. The bold numbers indicate how many counters we found in each of the architecture blocks. 66

6.6 Solution as subgraph of the design graph for the write pointer counter in the FIFO of the *sasc* design. . . 66

6.7 Counter structure from the *i2cslave* design. The search operation finds two counters, as two multiplexer-chain paths connect the $add and $dff node. Post-processing merges this to one counter. 67

[4] *OpenCores project "USB 2.0 Function Core"*. https://opencores.org/projects/usb. Accessed: 2019-11-26.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.8 Branch merging performed by the post-processing stage in the design *fast_log*. Each of the two branches connected to the $pmux multiplexer are a distinct result from the search operation. Post-processing can merge them to comparison tree. 70

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Acronyms

AES Advanced encryption standard
ALU Arithmetic logic unit
API Application programming interface
AST Abstract syntax tree
BFS Breadth first search
BNF Backus–Naur form
CSP Constraint satisfaction problem
CSS Cascading style sheets
DFA Deterministic finite automata
DFS Depth first search
DMA Direct memory access
EDA Electronic design automation
FIFO First in first out
FPGA Field programmable gate array
FSM Finite state machines
GT Generate-and-test
HDL Hardware description language
IC Integrated circuit
IP Intellectual property
LFSR Linear-feedback shift register
LSB Least significant bit
LUT Lookup table
MSB Most significant bit
NFA Nondeterministic finite automata
PGSL Pattern graph specification language
PHY Physical layer
PL Protocol layer
PRNG Pseudo random number generator

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

QBF Quantified boolean formula

RAM Random access memory

RELIC Reverse engineering logic identification and classification

RTL Register transfer level

RTLIL RTL intermediate language

SMT Satisfiability modulo theories

SoC System on chip

SSRAM Synchronous static RAM

UART Universal asynchronous receiver transmitter

USB Universal serial bus

UTMI USB transceiver macrocell interface

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 1

Introduction

This thesis uses the concepts *design graph* and *pattern graph specification language (PGSL)* which were introduced in the PhD thesis of Christian Krieg.[a] In this thesis we use these concepts, but with different terminology for their elements. A detailed mapping between the two terminologies is provided in Table 1.1. The reason for using a different terminology is to maintain consistency with the implementation of the search algorithm, which is the the main contribution of this thesis. If you want to refer to concepts of PGSL and the underlying graph model, please use the terminology given in the PhD thesis of Christian Krieg.

[a] C. Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

1.1 Problem statement and proposed solution

Design engineers use an hardware description language (HDL) like *Verilog* to design hardware descriptions that get synthesized to circuits with the help of electronic design automation (EDA) tools. Subsequently, verification engineers check these designs against their specification and look for errors. A new, emerging task of verification engineers is to also perform security evaluations of designs. A deep understanding of the design is needed to perform these tasks. Besides simulation, analyzing the HDL description is a common method for design understanding. Nevertheless, HDL descriptions are prone to coding styles and the same functionality can be described in multiple ways. Reverse engineering and examining structures of the synthesized design offer a complementary approach to gain deep understanding of a design.

In this thesis, we present a methodology for searching substructures in a given HDL design. Our methodology enables a verification engineer to both formulate arbitrary structures that are of interest to him and perform automated search in a given design. In the context of high-level structures we call these structures *functional primitives*, an example being a counter. The key parts of our methodology are: (1) a search-space abstraction called *design graph*, (2) search patterns modeled using the pattern graph specification language (PGSL) and (3) a search algorithm that searches for the patterns in the design graphs. The main focus of this thesis is on part (3), the development of the search algorithm. To be conveniently usable by a verification- or reverse engineer, we integrate our entire flow in the open-source synthesis

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 1.1: Example module for a FIFO (adapted from the OpenCores *sasc* core[2]). The read and write-pointers are incremented in line 20 and 25, these operations implement the counting characteristic of the counters.

```
1 module fifo4(clk, rst, clr, din, we, dout, re, full, empty);
2
3 input         clk, rst, clr;
4 input   [7:0] din;
5 input         we, re;
6 output [7:0]  dout;
7 output        full, empty;
8
9 // Local Wires
10 reg   [7:0] mem[0:3];
11 reg   [1:0] wp;
12 reg   [1:0] rp;
13 wire        full, empty;
14 reg         gb;
15
16 //Read & Write Pointers
17 always @(posedge clk or negedge rst)
18     if(!rst)     wp <= 2'h0;
19     else if(clr) wp <= 2'h0;
20     else if(we)  wp <= wp + 2'h1;
21
22 always @(posedge clk or negedge rst)
23     if(!rst)     rp <= 2'h0;
24     else if(clr) rp <= 2'h0;
25     else if(re)  rp <= rp + 2'h1;
26
27 // Fifo Output
28 assign  dout = mem[ rp ];
29
30 // Fifo Input
31 always @(posedge clk)
32     if(we) mem[ wp ] <= din;
33
34 // Status
35 assign empty = (wp == rp) & !gb;
36 assign full  = (wp == rp) &  gb;
37
38 // Guard Bit
39 always @(posedge clk)
40     if(!rst) gb <= 1'b0;
41     else if(clr)  gb <= 1'b0;
42     else if((wp + 2'h1 == rp) & we) gb <= 1'b1;
43     else if(re) gb <= 1'b0;
44
45 endmodule
```

suite *Yosys*.[1] This way it is accessible from the synthesis tool.

To make the related work easier to understand for readers, the following paragraphs outline the key points of the search methodology presented in this thesis. We illustrate our search methodology with an example search of the functional primitive counter in a *Verilog* module that implements a first in first out (FIFO) buffer. Extracting functional primitives supports reverse- and verification engineers in checking a given design against its specification, identify errors, perform security evaluations, and deepen overall understanding of a design.

A FIFO allows writing to and reading from a buffer. Write and read pointers store the current positions of the buffer which should be read from or written to next. On each read or write operation the related pointer is incremented. These incrementation operations implement counters. An example *Verilog* module for such a FIFO is shown in Listing 1.1. Specifically, the lines 20 and 25 implement the characteristics of a counter.

As a preparation for the search operation, we abstract target HDL designs to eliminate coding style, enable permutation-independent identification of functional primitives and reduce overall search space complexity. To remain at a high-level abstraction, we first perform high-level synthesis using *Yosys*. We use the resulting *Yosys* internal RTL intermediate language (RTLIL), a bipartite graph at register transfer level (RTL), as starting point for further abstraction.

[1] C. Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

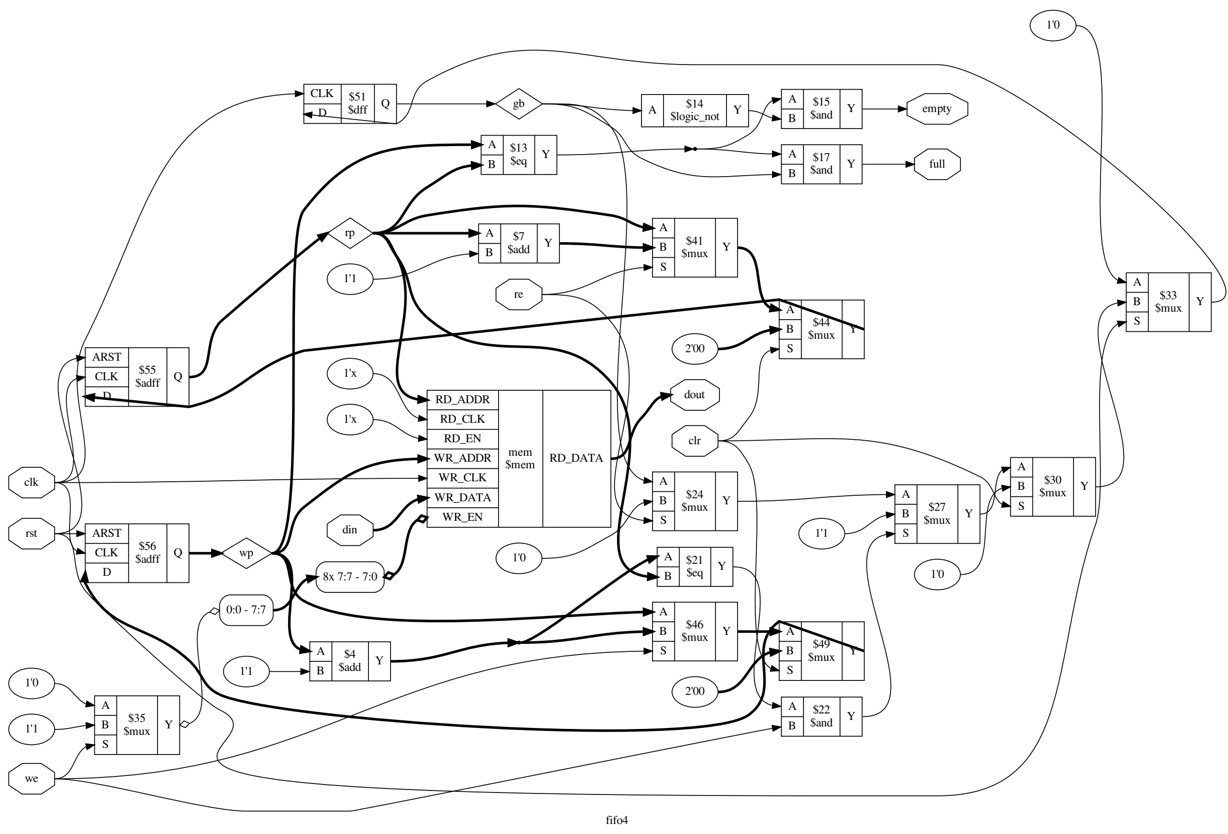


Figure 1.1: The RTLIL in graph representation of the example FIFO. At RTLIL the module is synthesized and contains cells with ports and nets connecting ports.

RTLIL represents a netlist with cells, cell ports and nets. The RTLIL for our example FIFO can be seen in Figure 1.1.

We further abstract to an abstract graph model which we call *design graph* using a custom *Yosis* plugin which was developed and implemented by Christian Krieg.[3] The design graph is a monopartite graph and abstracts away graph information which we don't need for our pattern search. Specifically, we omit cell ports and nets. This graph model contains

(a) nodes that are labeled with the type of the cell from the RTLIL they represent,

(b) nodes that represent constants,

(c) nodes that represent primary inputs, and

(d) nodes that represent primary outputs.

The example FIFO abstracted as design graph is shown in Figure 1.2.

To model patterns we want to find in a design, we use the pattern graph specification language PGSL.[4] PGSL is based on the observation that functional similar structures often vary in the duplication of subparts or bit widths. Its building blocks are the same elements, that can be found in the design graph. Again we want to emphasize that the terminology concerning PGSL differs from the terminology used by Krieg.[5] The terminology used in this thesis reflects the implementation of the developed search algorithm, a comparison of the two terminologies can be seen in Table 1.1.

3 Krieg, "Pattern-Based Hardware Trojan Characterization for Design Security Assessment".
4 Ibid.
5 Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

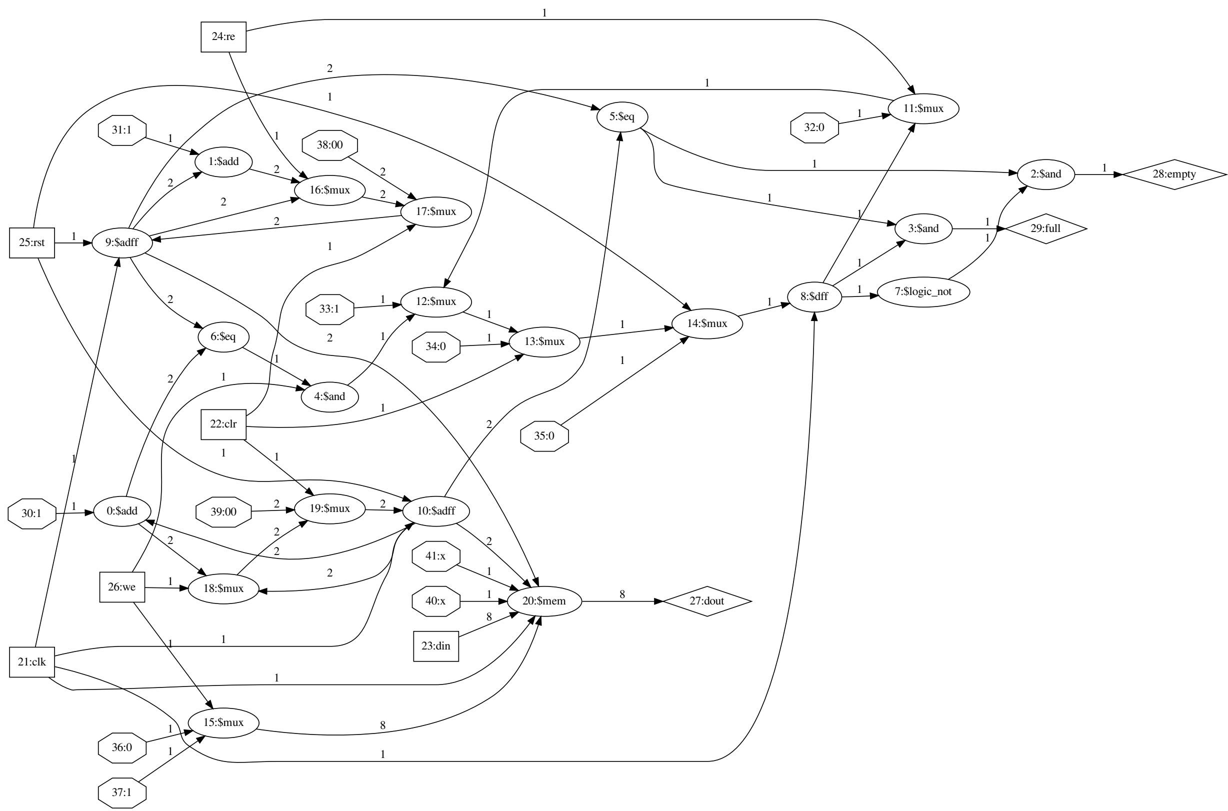


Figure 1.2: The design graph abstraction of the example FIFO. The result is a directed graph consisting of nodes with types and connections between these nodes.

Table 1.1: Comparison of the terminology concerning PGSL between this thesis and the PhD thesis of Christian Krieg.[6] The terminology used in this thesis reflects the implementation of the developed search algorithm.

This thesis	PhD thesis of Christian Krieg
Single node	Trivial block
Composite node without quantification	Unquantified block
Composite node with quantification	Quantified block

The basic element of a PGSL pattern is a *single node* (e.g., $and for an AND node) which matches one node of a specified type in the design graph. In this context strings in a PGSL pattern prefixed by the character '$' are references to internal cell types of *Yosys*. A single node can also be specified to match one node in a set of node types, for example $dff | $dlatch matches data flip flops or data latches. Nodes in PGSL can be connected serially (operator '->') or in parallel (operator ' || '). Multiple nodes can be grouped (using parentheses) to form *composite nodes*. The bit width of a connection is not specified to keep the patterns general. Composite nodes can be elements of a composite node. Therefore, a PGSL pattern forms a hierarchy of subpatterns.

To account for structural variation, we introduce quantifications that are inspired by regular expressions. A node in PGSL is quantified by appending a quantification to its specification. For example, the quantification >+ indicates that the preceding node should be matched serially one or more times.

[6] C. Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

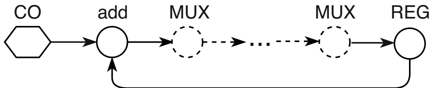


Figure 1.3: Graph view of the PGSL pattern for counters.

Using PGSL we can for example model a pattern for the structural kernel of counters:

\CO -> [[$add -> @MUX >*-> @REG]] ; (1.1)

This pattern specifies the structural kernel of counters as a register memory-element (@REG) that is incremented by a constant value (\CO) with the help of an addition ($add). The path between the addition node and the register can contain a multiplexer-node chain (@MUX >*) that implements the counter's control path. Finally, the loop-grouping operators '[[' and ']]' specify that the the right side of the grouping (in our case @REG) should have a connection to the left side of the grouping (in our case $add). This loop connection specifies our desired feedback from the register to the addition. The pattern termination character ';' specifies the end of a pattern. The PGSL elements @REG and @MUX are macros which are expanded by our PGSL parser. @REG expands the type specification $dlatch | $dff | $adff and @REG to $mux | $pmux. A graph view of this pattern is shown in Figure 1.3.

Our search algorithm uses the design graph of a given *Verilog* HDL design and a PGSL pattern to perform a subgraph isomorphism search. As a pattern contains subpatterns, which themselves can contain subpatterns, our patterns are hierarchical. Therefore, we design our search to be hierarchical. We solve from bottom to top by combining subpattern matches to subpattern matches one level higher in the hierarchy. We search for subpattern matches using a constraint satisfaction problem (CSP) approach. A CSP has a set of variables which each have a set of possible values we can assign to them ("domain") and a set of constraints on these variables. The variables of our CSP are the nodes contained in a subpattern. Each node has candidates which are nodes or subgraphs of the design graph. Connections between the nodes of our subpattern form constraints on the nodes.

To solve the CSP we use backtracking search, a depth-first tree-search-strategy. Using backtracking search, In succession we assign each node of our subpattern one of its candidates. After each assignment, we check if any constraint is violated. If no constraint is violated, the next unassigned node is assigned one of its candidates. If a constraint violation is found we backtrack. Backtracking unsassigns the last assigned node and continues the search process with its next candidate. A complete assignment where every node has one of its candidate assigned without any constraint violated is a subpattern match. Backtracking also occurs when a subpattern match was found, or a node has no more candidates to assign.

Figure 1.4 gives a high-level overview of the architecture we propose for our pattern search methodology. Input to our search methodology is on one side a *Verilog* HDL design and a search pattern modeled in PGSL on the other side. The HDL design is first synthesized using high level synthesis offered by *Yosys*. Subsequently, the synthesized design is converted to a design graph. The search pattern is parsed to a representation we call *pattern graph*. We use the pattern graph to filter the design graph to a filtered design graph that only contains types that appear in the pattern graph. This reduces the overall size of the search space for our pattern search. The filtered design graph and the pattern graph

[7] C. Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

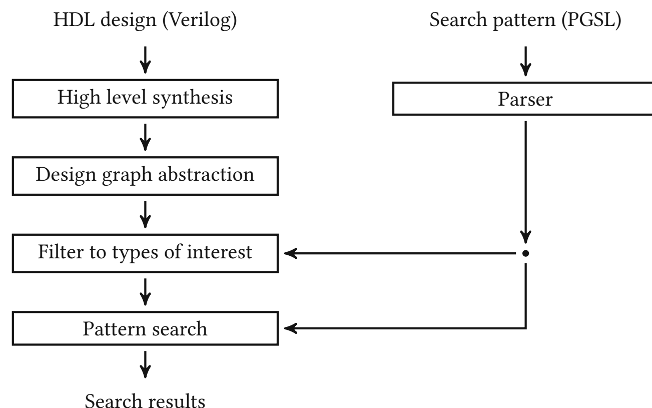


Figure 1.4: Overview of the architecture proposed in this thesis. A *Verilog* HDL design is synthesized using *Yosys* and abstracted to a design graph. Search patterns modeled in PGSL are run through a parser and used to first filter the design to "types of interest" i.e. the types the pattern contains. This pattern and the filtered graph are used together in a pattern searcher, that constructs search results, that match the pattern in the design graph. A similar architecture overview can also be found in the PhD thesis of Krieg[7] as that work also discusses the algorithm that was developed in this thesis.

are the inputs to the search algorithm that solves the subgraph isomorphism using a CSP solver bottom up, combining subpattern matches until the top-level pattern is matched. The result of a search operation is either empty, when no match was found, or a list of graphs that are subgraphs of the design graph. In the case of our example, the search for counters in the FIFO design, we find two subgraphs that implement the structural kernel of the read and write-pointer counters. In the design graph of the FIFO design (see Figure 1.2) these results are the subgraphs of nodes with ids $\{31, 1, 16, 17, 9\}$, and $\{30, 0, 18, 19, 10\}$.

The given example reflects the problem statement for this thesis: The development and implementation of a search algorithm that finds subcircuit patterns modelled with PGSL in design graphs of Verilog HDL designs.

1.2 Research questions and thesis structure

The main research questions based on the problem statement given in Section 1.1 are:

1. How can we find netlist structures modeled as serial-parallel graphs in a given *Verilog* HDL design?

   To answer this question, we first in Chapter 2 discuss the state of the art concerning reverse engineering and identifying high-level circuits in digital designs. We identify strengths and deficiencies of different approaches, and contextualize our methodology that is able to face the problem of structural variability in subcircuits with similar functionality.

   To reduce the search space complexity we use the design graph abstraction as target graphs for our pattern search. A description of this abstraction can be found in Section 3.1.

   In Chapter 5 we present our search methodology step by step and the chosen design decisions. The background needed to understand the basis of solving problems using a CSP approach is presented in Section 3.2.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

2. How can such graph-structure patterns be modeled?

   We model structural search patterns with the flexible, expressive pattern language PGSL. We use PGSL as it allows us to model patterns with structural variability. Chapter 4 describes the elements of this language, the reasoning behind its constructs, how they work together and how we use it to formulate the patterns we use in our experiments. We demonstrate the effectiveness of our chosen pattern modelling language PGSL in Section 4.2 by developing patterns for the following functional primitives: counters, state machines and elements of decoders/encoders.

3. How can a search be done effectively and efficiently?

   In Chapter 6, in order to evaluate our search methodology, we conduct a large scale experiment with 5 selected PGSL patterns and 74 open-source HDL designs. We classify both designs and patterns with metrics regarding the graphs' structure and cell type composition and present performance measurements for the respective searches. The results clearly show practical feasibility, and very acceptable runtimes in the order of seconds to minutes.

Finally, in Chapter 8, we provide a summary of the answers to the research questions, present our major findings and identify potential topics open to future work.

1.3 Contributions

This thesis yields the following contributions:

1. Development of a search algorithm to find PGSL patterns in so called "design graphs" which are an abstraction of *Verilog* HDL designs that have been synthesized at RTL.

2. Implementation of the developed search algorithm as plugin for the open-source synthesis suite *Yosys* with a custom solver based on CSP solving methodology.

3. Validation of the efficiency and effectiveness of the developed search algorithm by means of a large scale experiment with 74 open-source HDL designs and five well selected search patterns for finding state machines, counters, and elements of encoders and decoders.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 2

Related work

This work presents a method to extract subcircuit patterns from a target graph by subgraph isomorphism. We abstract both pattern and target circuit as directed graphs, where labeled nodes represent operations, and edges represent information flow through these nodes. Patterns are specified using the pattern graph specification language (PGSL). A pattern consist of nodes and connections between these nodes. Nodes in a pattern represent: (a) a cell type in the target design, (b) a primary input of the target design, (c) a primary output of the target design, or (d) a constant in the target design. Subparts of the pattern can be structurally quantified using an approach similar to regular expressions. The subgraph matching method works hierarchically and uses constraint satisfaction problem (CSP) to formulate and solve the subgraph isomorphism problem. Our subgraph-matching method enables a verification engineer to extract high-level information from a target design by searching for patterns that he or she models using PGSL.

Reverse engineering digital circuits is the process of extracting high-level information from low-level circuits. This process analyzes low-level designs, identifies subcircuits that form functional primitives and ultimately helps a design engineer to understand the circuit as a whole on an algorithmic level. Extracted high-level blocks and knowledge can subsequently be used as input to subsequent design analyses. Potential examples are high-level verification[1] and identifying malicious functionalities in designs (hardware Trojan detection).[2] Last but not least, reverse engineering is also a way to produce higher abstraction implementations for designs for which only low-level netlists are available (e.g, synthesized intellectual property (IP) cores).

One of the early publications concerning reverse engineering by Hansen et al.[3] outlines reverse engineering techniques needed to create a high-level view of a given circuit. Hansen et al. analyze the ICSAS-85 benchmark set.[4] For these industrial designs neither function nor high-level designs had been published. The techniques Hansen et al. use to abstract to a high-level view, which in their case where mostly done manually, include:

[1] S. Kundu et al. “Translation Validation of High-Level Synthesis”. In: *High-Level Verification: Methods and Tools for Verification of System-Level Designs*. New York, NY: Springer New York, 2011, pp. 97–121. ISBN: 978-1-4419-9359-5. DOI: 10.1007/978-1-4419-9359-5_7. URL: https://doi.org/10.1007/978-1-4419-9359-5_7.

[2] C. Krieg. “Pattern-Based Hardware Trojan Characterization for Design Security Assessment”. PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

[3] M. C. Hansen et al. “Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering”. In: *IEEE Design Test of Computers* 16.3 (July 1999), pp. 72–80. ISSN: 0740-7475. DOI: 10.1109/54.785838.

[4] F. Brglez and H. Fujiwara. “A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran”. In: *Proceedings of IEEE Int’l Symposium Circuits and Systems (ISCAS 85)*. IEEE Press, Piscataway, N.J., 1985, pp. 677–692.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

(1) *Library modules*: By comparing parts of the circuit to known components from cell libraries and textbooks, subcircuits can be extracted.

(2) *Repeated modules*: Often identical subcircuits appear multiple times in a design, as they form a multi-channel or multi-bit high-level operation.

(3) *Expected global functions*: From the knowledge which functionalities parts of a circuit fulfill, the global functionality can be expected.

(4) *Computed functions*: Combinatorial subcircuits can be viewed as their high-level logic function in symbolic or binary (truth table) form.

(5) *Control functions*: Key control signals can be identified to partition a complex function into simpler ones.

(6) *Bus structures*: Bit signals can be grouped to bit-array bus signals.

Our subgraph-matching method can be used for (1), (2), and (3) of the techniques the authors present. A reverse engineer can model library modules as patterns and subsequently search for them in a target design. He or she can model a pattern for a single channel of a multi-channel high-level operation. Our search method provides all the single-channel matches in the target design. With the knowledge of the location of the single-channel operations, the reverse engineer can locate the multi-channel operation. Finally, a reverse engineer can run multiple searches with patterns for which he knows the functionality, and conclude the overall functionality of a design.

In the following, we analyze the field of reverse engineering. Furthermore, we discuss different approaches for searching high-level structures in designs. We discuss PGSL in Chapter 4, the related work concerning the CSP in Section 3.2, and the related work concerning the subgraph isomorphism problem in Section 3.3.

2.1 Low-level reverse engineering

At the lowest level digital designs are unstructured, flattened gate-level or bit-level netlists. These designs are the result of logic synthesis and technology mapping. To gain understanding of low-level designs, reverse engineers use the following methods to promote the design to a higher level: (1) identifying words (aggregates of individual bits to bit-arrays), (2) structuring the design into control and data-logic, and (3) identifying simple operations on words such as addition, boolean logic, shift. In contrast, the subcircuit-matching method we present in this thesis takes a higher-level representation as input, in particular an abstraction of RTL netlists from which we extract algorithmic building blocks.

Li et al. propose *WordRev*[5] to extract word-level information from a given bit-level netlist. First, they group individual wires, that have the same backward reachability, or which are functionally equivalent. Next, they propagate these words across the netlist to reach as many other words as possible. The result is a word graph. The nodes of this word graph represent identified words and connections represent the reachability between words. Based on this graph, the authors cut out portions of the netlist between words. They formulate a boolean formula for this cutout section and check for functional equivalence to reference word operations (e.g., addition, boolean operation, shifting). Our approach does not cut out portions of a netlist, but rather looks at the design at a whole to find subcircuit pattern-matches.

[5] W. Li et al. "WordRev: Finding word-level structures in a sea of bit-level gates". In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. June 2013, pp. 67–74. DOI: 10.1109/HST.2013.6581568.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Tashjian and Davoodi[6] group bits to words by structural analysis. They identify similar nets and subtrees which have similar structure and are close to each other. Additionally, they identify control signals. Control signals tend to affect word structures and therefore aid in the search of words. The result is a word-level abstraction with affecting control signals which they conclude is an input for follow up analyses. Our approach does not look at individual nets, we abstract nets between two nodes in a design to a connection between those two nodes. This way our matching method does not need to consider the different permutations of nets.

In contrast, Yu and Ciesielski[7] use an algebraic approach to create a word-level abstraction and find word-level operations. First, they generate an algebraic equation for primary inputs to primary outputs by using an algebraic model of the design's internal gates. By iteratively rewriting and simplifying the equation, intermediate terms are created which form cuts in the design. These intermediate expressions are checked for subexpressions which form words. Having found words, the authors can identify integer arithmetic operations (e.g., addition, multiplication, shift) on those words. We do not use an algebraic approach to classify the functionality of a subcircuit. Our approach uses structural and node-type matching to specify a functionality we want to find in the target design.

Meade et al.[8] propose a low-level reverse engineering approach to categorize registers into data registers and control-logic registers. The authors state, that it is important to differentiate between data registers and control-logic registers to understand an unknown design. Their topology-based computational method reverse engineering logic identification and classification (RELIC) examines the logical and topological similarities between pairs of registers in a netlist. They assign each register pair a similarity score based on how similar their fan in logic is. Register pairs with high similarity scores are classified as being on the same data path, the remaining registers are classified as control-logic registers. With our pattern-matching method we can model patterns containing registers. For example a node *($dff | $adff)* matches to a register that is either a synchronous or an asynchronous data-flip-flop. Nevertheless, our method does not differentiate between data and control logic, as we use the information of subcircuits, that match to known patterns to establish the functionality of an unknown design.

2.2 Finding high-level components

In order to understand the algorithmic function of a design, one possibility is to identify high-level components in a design. For example, the knowledge, that a portion of a design constitutes a counter is more desirable, than just knowing, that the design contains registers and addition cells. Therefore another field of research in reverse engineering focuses on identifying high-level components by comparing the given design to library components for which we know the algorithmic functionality. This comparison can either be done based on checking functional equivalence or structural equivalence. Additionally, hybrid approaches exist that try to combine both approaches.

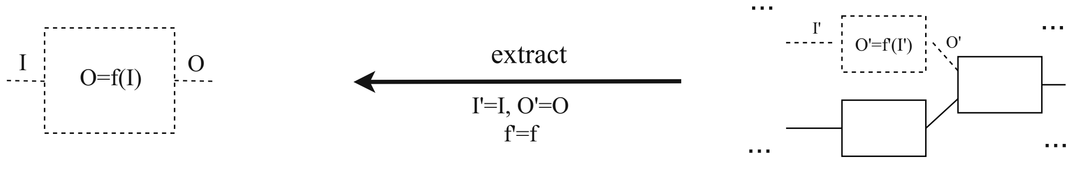
Functional matching and structural matching each have strengths and challenges. Functional-equivalence checking methods can disregard the structural variability of components with similar functionality since they only look at the relation between input and output values. This leaves the challenge of how to capture and compare the characteristic

[6] E. Tashjian and A. Davoodi. "On Using Control Signals for Word-level Identification in a Gate-level Netlist". In: *Proceedings of the 52Nd Annual Design Automation Conference*. DAC '15. San Francisco, California: ACM, 2015, 78:1–78:6. ISBN: 978-1-4503-3520-1. DOI: 10.1145/2744769.2744878.

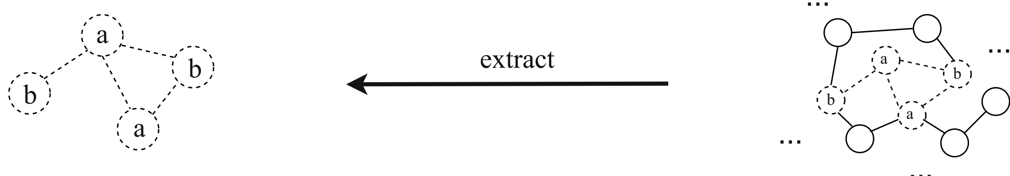
[7] C. Yu and M. Ciesielski. "Automatic word-level abstraction of datapath". In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. May 2016, pp. 1718–1721. DOI: 10.1109/ISCAS.2016.7538899.

[8] T. Meade et al. "Gate-level netlist reverse engineering for hardware security: Control logic register identification". In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. May 2016, pp. 1334–1337. DOI: 10.1109/ISCAS.2016.7527495.

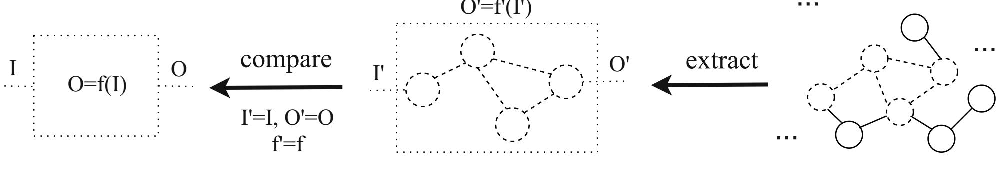
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, WIEN Your knowledge hub



(a) Functional matching: Gascón et al. (2014), Soeken et al. (2015), Li et al. (2012)



(b) Structural matching: This work, Ohlrich et al. (1993), Olbrich et al. (2001), Cakir et al. (2018), Terem et al. (2004)



(c) Hybrid matching: Subramanyan et al. (2014), Shi et al. (2012)

Figure 2.1: Different approaches to find high-level components in designs and the related work that we will discuss. Functional matching finds subcircuits whose function matches a component in a library. Structural matching checks for subgraph isomorphism. Hybrid matching combines both methods in multi-step approaches.

behavior as relation of inputs and outputs of subcircuits and reference components. Structural methods do not capture characteristic behavior as a relation between inputs and outputs. Instead, they compare the structure of a subcircuit with a reference circuit. The structure of a component is easier to capture and compare, however this leaves the method prone to problems with modelling structural variability in reference components, in order to not have multiple reference components for components with the same functional core. Hybrid approaches use a multi-step approach to combine structural and functional equivalence checking. The search is split into multiple algorithms, each searching for a specified class of components. This approach can be undesirable, as for every new component class, a new search algorithm has to be developed.

Figure 2.1 gives an overview over these different approaches. The subcircuit-pattern-matching method proposed in this thesis is purely based on structural equivalence checking.

Functional Matching

Gascón et al.[9] present an algorithm to extract functional blocks that are functionally equivalent to a given template. Users can specify template patterns with a functional specification that applies word-level operations such as concatenation, extraction and algebraic operations on inputs to produce outputs. These templates are formulated with an extension of the *Yices* language.[10] The authors capture functionality with boolean formulas and check for equivalence with a satisfiability modulo theories (SMT) solver. Their idea to use a description language is similar to our approach with

[9] A. Gascón et al. "Template-based Circuit Understanding". In: *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design.* FMCAD '14. Lausanne, Switzerland: FMCAD Inc, 2014, 17:83–17:90. ISBN: 978-0-9835678-4-4.

[10] B. Dutertre. *Yices Manual.* July 28, 2016.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

PGSL. However, their description formulation is not agnostic to word width. Therefore they end up having a library with modules that are functionally equivalent, but differ in bit width. PGSL by design does not care about word widths and structures can be quantified to account for different input and output word widths.

Soeken et al.[11] identify components from a component library in a given design with the help of so called simulation graphs. They state that many components have an unique input-output characteristic that can be captured from small sets of input vectors. They call these input vectors simulation vectors and capture the behavior of a subcircuit with a so called simulation graph. Simulation graphs connect inputs that are logic *1* for a simulation vector to outputs that are logic *1* for this input simulation vector. They create simulation graphs as labeled graphs for both the library components and the target circuit. A subgraph isomorphism approach is used to compare candidates against library components. The authors do not check for structural equivalence of the subcircuit and the library components. They rather compare the simulation graphs that are based on the behavior. The quality of candidates and simulation graphs depends on the simulation vectors used. The authors categorize their simulation vectors by using k-hot and k-cold categories. For k-hot, k is the number of logical *ones* in a simulation vector. Similarly, k-cold are simulation vectors with just k logical *ones* and the rest logical *zeroes*. Exhaustive simulation is needed to identify the appropriate set of simulation vectors for different library components. The authors state that most of their experiential library components can be fully specified with 0-cold, 1-hot and 2-hot simulation vectors, but a multiplier needs a different set of simulation vectors (2-hot, 0-cold and 1-cold vectors). Therefore, new components might need yet another set of simulation vectors to be fully specified. This implies two possibilities:

(1) a user has to find out which set of simulation vectors are needed to categorize a new component and pass this information to the search algorithm, or
(2) the search algorithm has to use all possible simulation vectors to be sure that all possible components can found.

Possibility (1) has the disadvantage that the user has to do initial simulation for a new component. With our method users can model search patterns in PGSL and immediately run searches in the target design. On the other hand, possibility (2) has the disadvantage that more simulation vectors imply bigger simulation graphs, which imply a bigger search space. With our search method the size of search space is only dependent on the size of the target circuit and the amount of different node types used in the pattern.

Similarly, Li et al.[12] use input-output traces to functionally characterize the behavior of subcircuits. First, the authors partition unknown circuits into subcircuits for which they mine input-output traces to gain properties for delta events of signals. An example for a delta event is a transition of a signal x from logic *0* to logic *1* that always infers another signal y to also transition from logic *0* to logic *1*. These properties are captured as a graph. The given example yields a vertex labeled $\Delta_{x_{0\to1}}$ connected to another vertex labeled $\Delta_{y_{0\to1}}$. Next, the authors compare the resulting graph against abstract library components for which the graphs are known, by finding maximum common subgraphs. The result is a mapping of input and output signals to the signals of an abstract library component. Finally, the authors use model checking to verify that the subcircuit is an instance of the abstract library component. Overall, in order for a subcircuit to be a match to an abstract library component, it has to have a match for all the input and outputs defined by the component. This implies that even if the unknown subcircuit's functionality is the same as the component, it might not

[11] M. Soeken et al. "Simulation Graphs for Reverse Engineering". In: *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*. FMCAD '15. Austin, Texas: FMCAD Inc, 2015, pp. 152–159. ISBN: 978-0-9835678-5-1.

[12] W. Li et al. "Reverse engineering circuits using behavioral pattern mining". In: *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*. June 2012, pp. 83–88. DOI: 10.1109/HST.2012.6224325.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

be registered as a match. Our approach is more precise in this matter. It does not rely on input and output matching, but on matching of inner connections and node types. If the nodes and the connection between them match, we imply that the functionality is the same.

Structural Matching

An early example of purely structural matching is SubGemini[13] by Ohlrich et al. Their method finds a given circuit as subcircuit in a target circuit using a labeling algorithm and breadth first search (BFS). Their approach has two phases. Phase one identifies all possible locations of the circuit in the target circuit. First, they identify a key vertex in the circuit that they intend to search for. In a second step vertices in the target circuit are identified which match the key vertex. In the second phase, starting from the key vertex candidate, they iteratively label the surrounding vertices in both subcircuit and target circuit. If all labels from the search circuit are found in the target circuit a subcircuit match is found. Compared to our approach, their graphs have distinct vertices for nets (wires) that are labeled by their width. With PGSL, we abstract nets to connections between vertices and no bit width is specified. This makes the matching more generally applicable. Additionally, our search can not use BFS, as we do not work with fixed graphs but quantified graphs and use hierarchical search.

Olbrich et al.[14] use structural identification of subcircuits for circuit substitution in designs. Like us, they allow patterns to have subpatterns. In addition, they allow the treatment of optional and multiple ports. This can be seen as a form of quantification of structures. Nevertheless the quantification possibilities we have using PGSL are more extensive. PGSL allows the specification of subpatterns which number of occurrence can be quantified either in serial or in parallel.

Cakir and Malik[15] present a structural matching approach based on *geometric embedding*. Their method extracts designs given as HDL RTL code from a flattened gate-level netlist. With our methodology we also can transform a HDL description to a design graph and create a search pattern based on this graph. But in comparison, our search space is based on a higher level, the RTL netlist of a synthesized HDL design. For their search both the reference HDL design and the target flattened gate-level netlist are transformed to undirected graphs. In contrast, our patterns and our search space are directed graphs, where cells represent operations, and edges represent directed information flow through these cells. The first step of their geometric embedding is to compute affinity matrices for each vertex that hold the connections to k closest neighbours. Next, eigenvalues for these matrices are calculated. Finally, they use these eigenvalues as features to search for the reference circuit in the target circuit. Cakir and Malik clarify that their "tool is not a fine matching like a shift register or counter identification".[16] Their tool uses reference design information to identify blocks in the target circuit, it compares features not the structure of the circuits. Therefore it is possible that a reference circuit that is a part of the target design, might not be identified in the target design. Nevertheless, their experiments show an over 90% accuracy for identifying their searched high-level blocks in the target design. In contrast our matching, as it directly compares the structure of graphs, can assure that a submodule of a target design, which was transformed to a pattern, matches in the target design.

[13] M. Ohlrich et al. "SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm". In: *30th ACM/IEEE Design Automation Conference.* June 1993, pp. 31–37. DOI: 10.1145/157485.164556.
[14] M. Olbrich et al. "A Combined Structural and Symbolic Method for Automatic Behavioral Modeling of Nonlinear Analog Circuits". In: (Sept. 2).
[15] B. Cakir and S. Malik. "Reverse Engineering Digital ICs Through Geometric Embedding of Circuit Graphs". In: *ACM Trans. Des. Autom. Electron. Syst.* 23.4 (July 2018), 50:1–50:19. ISSN: 1084-4309. DOI: 10.1145/3193121.
[16] Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

The work of Terem et al.[17] is the most similar to our approach. They use a constrained-based algorithm to search for patterns in hierarchical high-level designs by subgraph isomorphism. As in our method, designs and patterns are directed graphs with design elements ("instances", comparable to node types of PGSL and the design graph) and interface signals ("pins", comparable to our definition of primary inputs and outputs). Like us, they filter the target graph to only contain types that appear in the search pattern to reduce the search space. Our search method additionally filters candidates in a second step based on the surrounding nodes in the pattern graph which decreases the search space even further. The authors' method does not offer structural quantification of subpatterns. For example, if a user wants to find structures containing serial chains of a node type, he or she would have to run multiple searches with patterns containing chains of different length (2 nodes chain, 3 node chain, ...). Our methodology allows to specify quantified subpatterns (the pattern for the given example is *(type)>{2,}* to match serially connected chains of size 2 or more).

Hybrid Matching

Subramanyan et al.[18] extract data-path structures from an unstructured netlist. The authors combine structural and functional analysis in multi-step algorithms. They differentiate between combinatorial and sequential components. For combinatorial identification, the authors use a multi step approach: (1) Bitslice identification, using cuts through the circuit to find nodes that match a function in a bitslice library (e.g., $f(a,b,c) = ab + bc + ca$ as a full adder carry function), (2) aggregation to multibit components, (3) word identification and propagation, and (4) module identification by matching against a component library (e.g, an 8-bit arithmetic logic unit (ALU)). For finding sequential components, the authors offer separate algorithms for counters, shift registers, random access memory (RAM) and multibit registers. The authors use structural analysis by comparing to a reference graph (e.g., certain latch topologies might form a counter) and subsequently functional comparisons to a component library. Rather than having multiple separate algorithms, we decide to have a general solver and put this variability into the *(more)* powerful pattern language PGSL.

Shi et al.[19] also present a multi-step process to get functional modules from gate-level netlist. These steps include state machine extraction and removal, partitioning with connected-components analysis on graphs to get combinatorial blocks, least significant bit (LSB) matching, multiplexer identification, and finding parallel signals. LSB matching is a feature that we can compare to finding quantified blocks with PGSL. With LSB matching the algorithm of Shi et al. searches for small predefined structures by comparing parts of the circuit functionally to a library. In a second step the structures get expanded towards word-level with logic depth search. The authors use a multi-step approach that uses both structural and functional equivalence, as they state that each method on their own has weaknesses. The weakness they state for structural matching is that a component library would need to include multiple variants that differ slightly (e.g., in bit width). On the other hand, the authors state that functional matching does not scale well for big reference components. Therefore using small component models that are width independent and in a second step expanding the structures combines the best of functional matching and structural matching. We approach these problems with patterns that are agnostic to bit widths and small subpatterns that can be found individually and can be combined to bigger structures using quantification.

[17] Z. Terem et al. "Pattern search in hierarchical high-level designs". In: *Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems, 2004. ICECS 2004.* IEEE. 2004, pp. 519–522.

[18] P. Subramanyan et al. "Reverse Engineering Digital Circuits Using Structural and Functional Analyses". In: *IEEE Transactions on Emerging Topics in Computing* 2.1 (Mar. 2014), pp. 63–80. ISSN: 2168-6750. DOI: 10.1109/TETC.2013.2294918.

[19] Y. Shi et al. "Extracting functional modules from flattened gate-level netlist". In: *2012 International Symposium on Communications and Information Technologies (ISCIT)*. Oct. 2012, pp. 538–543. DOI: 10.1109/ISCIT.2012.6380958.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.3 Summary

The related work can be roughly divided into low-level reverse engineering methods and methods that intend to find high-level components in digital circuits. The topic of low-level reverse engineering deals with promoting the designs to a higher abstraction level. We cover this field only on the surface as our search operates on *Verilog* HDL designs, that we abstract to a graph structure we call design graph. (see Section 3.1). Nevertheless, our work could be used on top of the word-level abstractions to search for high-level functional primitives. Methods in this field are (1) aggregating bits to words, by finding similar nets or subtrees, (2) identifying simple word-level operations (e.g., add, shift, logical or), and (3) categorizing registers to structure the circuit. Methods for finding high-level structures in digital circuits compare subcircuits to library components. We divide the approaches into searching for functional equivalence, structural equivalence or mixed approaches that use both methods in a multi-step approach.

Functional approaches struggle with capturing the full and characteristic functionality of subcircuits to compare them to reference functionality descriptions. Structural methods struggle with capturing and identifying structural variability of components with similar functionality. This leads to multiple reference components, with similar functionality but slightly different structure. In addition, scalability for big designs can be a problem. Hybrid approaches aim at combing the strength and weaknesses of both approaches, but end up with multiple algorithms for different classes of components. Our approach mitigates the given problems with: (1) Structural matching, but on an abstraction of the design, that simplifies the search space by abstracting to a monopartite graph, omitting cell ports and nets, (2) a powerful unified structural search methodology based on a CSP solver with search space reduction by filtering at runtime, and (3) a intuitive pattern specification language PGSL that allows a user to specify structural variability.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 3

Background

In this thesis, we use patterns to find subcircuits in an abstraction of hardware description language (HDL) designs. We we call this abstraction *design graph*. The search for subcircuits is an instance of the *subgraph isomorphism problem* and we model the matching problem of subpatterns as *constraint satisfaction problem (CSP)s*. Furthermore, the pattern graph specification language (PGSL) which we use to model patterns is inspired by *regular expressions*. In this chapter we present and discuss selected topics to provide background knowledge about the methods we use in our search methodology.

3.1 Design graph abstraction

A design graph is an abstract graph that represents an HDL in which we want to find a pattern that represents a functional primitive. The design graph abstraction model was envisioned by Krieg[1] as a result of the observation that "directly searching HDL descriptions entails the problem that the same functionality can be specified in arbitrary fashion".[2] This is due to the nature of HDL specification. Hardware specification using an HDL offers a designer the freedom to describe the same functionality with different coding styles, naming conventions, and multiple language elements . In comparison, a netlist is an unambiguous and uniform representation of a synthesized HDL design.

Netlists are descriptions of connectivity of components in a circuit. The design graph abstraction uses register transfer level (RTL) netlists as its basis. This is a good fit for our pattern search as we want to find high-level functional primitives. In RTL netlists, components are called *cells* and the connection elements are called *wires*. In addition to these elements, RTL netlists can contain *numeric constants*. Each cell represents an operation and has a specific type such as *mux*, *dff*, and *add*. Furthermore, each cell has one or more *input ports* and *output ports*. Wires are used to connect cells to cells and numeric constants to cells. Each port consist of a set of individual port bits and each wire consists of a set of individual port bits. Each port bit and each bit of a numeric constant can connect to a bit of a wire.

Althougg RTL netlists are well suited for our pattern search, they still contain information that we do not need. We model our patterns for functional primitives as directed graphs with interconnected cells without regard to ports. Here is where

[1] C. Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.
[2] Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

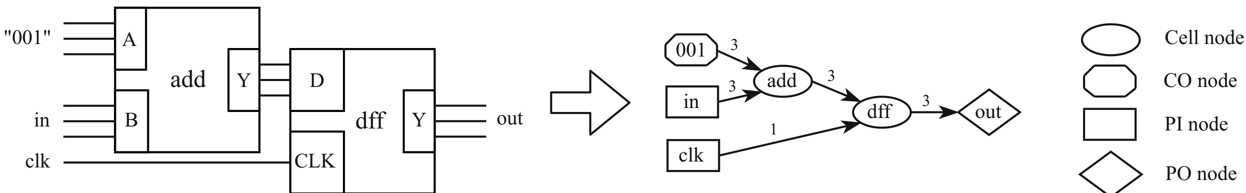


Figure 3.1: Visualization of an example model transformation from RTL netlist to design graph abstraction. With this abstraction input and output ports are omitted and individual wires are merged to directed edges.

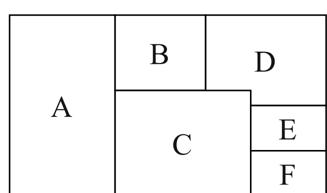
the design graph abstraction comes into play. It abstracts cells from an RTL netlist to nodes without ports and wires to directed edges. Furthermore, the design graph abstraction creates nodes for numerical constants, primary inputs, and primary outputs of RTL netlists. In sum, a design graph contains four types of nodes: (1) cell nodes, (2) primary input (PI) nodes, (3) primary input (PO) nodes, and (4) constant (CO) nodes. As edges in the design graph are directed, we do not lose information concerning the signal-flow directions in a design. The input-to-output and output-to-input information which is explicit in the RTL netlist is implicitly modeled in the direction of an edge. As a beneficial side effect of eliminating ports and introducing directed edges, two nodes in the design graph can at maximum be connected by one edge in each direction.

Figure 3.1 visualizes the model transformation from RTL netlist to design graph for an example circuit. The circuit adds binary one to a primary input *in* and stores the result in a clocked data flip-flop. The stored result is consequently a primary output *out*. Each of the cells of the netlist has distinct input and output ports and each wire represents one bit. Through the design graph abstraction, the ports are abstracted away and the bundles of wires are abstracted to directed edges with their bit width as labels. In addition, the primary input, primary output, and the constants are abstracted to own nodes.

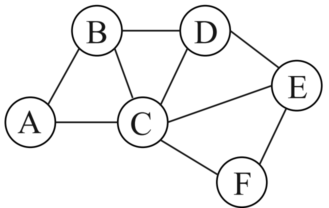
Krieg designed and implemented[3] the design graph abstraction model and transformation as so-called *pass* in *Yosys*.[4] When using *Yosys* for high-level synthesis of HDL designs it internally produces a representation of the synthesized design in RTL intermediate language (RTLIL) format. RTLIL is a netlist representation at RTL with a set of built-in cell types (indicated by a leading $). The model-transformation pass takes RTLIL as input and produces an C++ object oriented structure representing a design graph. This structure is accessible to subsequent *Yosys* passes. It contains objects of classes that represent the different node types in the design graph and an adjacency matrix. Each node has a unique ID and node objects are accessible via *C++ std::map* structures. The adjacency matrix of a design graph is of size NxN with N being the number of nodes in the design graph. This matrix stores how nodes are interconnected in the design graph. A non-zero element in a row *n* at position *m* indicates a directed connection from the node with ID *n* to the node with ID *m*. The value of the non-zero element indicates the bit width of the connection. In general, the adjacency matrix of a design graph is a sparse matrix (containing mostly zeros) as each node in the graph is only connected to a view of all available nodes. Therefore to maintain memory efficiency even for big designs, the adjacency matrix is stored as sparse matrix using the *Eigen C++ template library for linear algebra* in compressed-row format. This sparse matrix structure is a vector of adjacency lists. An adjacency list is a linked list that only stores non-zero elements of a row in the sparse matrix. For further implementation details refer to Krieg.[5]

[3] Krieg, "Pattern-Based Hardware Trojan Characterization for Design Security Assessment".
[4] C. Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/.
[5] Krieg, "Pattern-Based Hardware Trojan Characterization for Design Security Assessment".

TU Bibliothek, Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
WIEN Your knowledge hub The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Map coloring-problem



(b) Constraint graph

Figure 3.2: Visualization of an example map-coloring problem. Figure 3.2a is a fictional map with seven territories labelled A to F. In map coloring we have to color each territory with a color from a given set, such that no adjacent territories have the same color. Figure 3.2b is a visualization of the associated constraint graph. Each vertex represents one of the territories and each edge represents a coloring constraint between two adjacent territories.

3.2 Constraint satisfaction problem

The constraint satisfaction problem (CSP) is a very common problem representation for solving decision problems in which a set of decisions must be made and decisions interact with one another. It is widely used in artificial intelligence, other areas of computer science (e.g., scheduling, graph problems, circuit design), and operations research (e.g., financial engineering, supply chain management). With CSP we define a problem as a set of *variables* that need to be assigned *values*. Each variable has a set of possible values called a *domain*. Assigning a variable a value is a *decision*. Decisions can interact with each other, therefore not all possible combinations of decisions are valid. The interaction and compatibility of decisions are captured in a set of constraints.[6] We give the definition for the simplest CSP in Definition 3.1.

Definition 3.1 (Simplest CSP[7]):
The simplest kind of CSP consists of:

1. *a finite set of variables* $X = \{x_1, ..., x_n\}$,
2. *a finite and discrete domain* D_i *of possible values for every variable* $x_i \in X$, *and*
3. *a finite set* $C = \{C_1, ..., C_m\}$ *of constraints on the variables of* X.

A CSP has a state space. A *state* of a CSP is an *assignment* of values for some or all variables, $\{X_i = v_i, X_j = v_j, ...\}$. An assignment is *consistent*, if it violates no constraint. A *complete assignment* is an assignment in which every variable is assigned a value from its domain. Finally, a solution of a CSP is a assignment that is consistent and complete.[8]

The classic example for elaborating the concepts and solving of CSPs is the map-coloring problem. In this problem we are given a set of territories in a map like shown on in Figure 3.2a. We label these territories from A to F. In the map coloring problem, we need to color each territory of the map by a color from a set of colors, for example *red*, *green*, and *blue* (we abbreviate them as r, g, and b). The coloring has to be done such that no two adjacent territories in the map have the same color. Translating this to a CSP gives us:

1. 6 variables: $X = \{A, B, C, D, E, F\}$
2. The domain for each variable $D_i = \{r, g, b\}$ with $i \in \{A, B, C, D, E, F\}$
3. 8 constraints specifying that adjacent territories can not have the same color:
   $C = \{A \neq B, A \neq C, B \neq D, C \neq D, D \neq E, C \neq E, E \neq F, C \neq F\}$

[6] V. Kumar. "Algorithms for Constraint Satisfaction Problems: A Survey". In: *A.I. Mag* 13 (Oct. 1998).
[7] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009, pp. 202–239. ISBN: 0136042597.
[8] Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

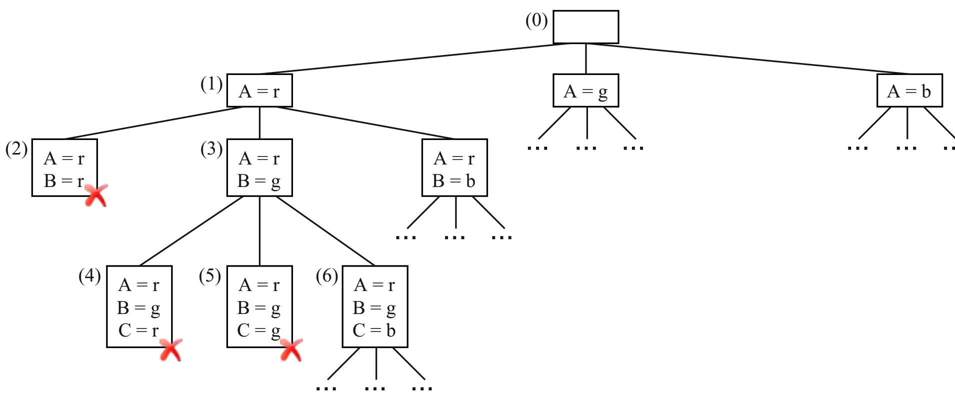


Figure 3.3: Backtracking search tree for our example map-coloring problem. The numbers indicate the traversal of the tree. Assignments marked with X violate a constraint and cause a backtrack.

The variables and the constraints can be visualized in a constraint graph as can be seen in Figure 3.2b. Each vertex represents one of the territories and each edge represents a color constraint between two territories.

A naive approach to solve a CSP is to use the generate-and-test paradigm.[9] Using this paradigm, each possible complete assignment of the CSP is generated and discarded if it violates a constraint. The remaining complete assignments are solutions for the CSP. The number of combinations to consider by using this paradigm is the Cartesian product of all the variable domains (in our case $3^6 = 729$).

A more efficient method is to use *backtracking search*. Backtracking search is essentially a depth first search (DFS) through the state space of an CSP that abandons a branch after determining that no solution is possible further down the branch. In this search strategy we instantiate variables one after another. A variable is instantiated by assigning it one of the values from its domain. Using backtracking search we incrementally extend an assignment. After each instantiation of a variable, the constraints concerning this variable are checked. If a constraint is violated, a backtrack occurs and we assign the last instantiated variable another value from its domain (e.g., the next value in the domain). We also backtrack if a variable has no more values to assign. This backtracking at violation of constraints eliminates subtrees of the search and therefore is more efficient than using the simple generate-and-test paradigm. If after an instantiation no constraint is violated, we traverse further down in the search tree by instantiating the next unassigned variable (e.g., the next variable in the ordered set $\{A, B, C, D, E, F\}$). If we are only interested in one solution for the CSP, the backtracking search aborts as soon as the first complete and consistent assignment has been found. Otherwise, we note the solution and backtrack to find all solutions.[10]

A part of this tree traversal for our example map-coloring problem is visualized in Figure 3.3. We start with an empty assignment at (0). At (1) we instantiate the first variable A by assigning it the value r (*red*) which is the first value of the domain of A. This assignment is consistent, as no other variable has been assigned a value yet. Next, we instantiate B at (2) and again assign the first value from its domain which is also r (*red*). The assignment $(A = r, B = r)$ violates the constraint $A \neq B$. Therefore, we backtrack and assign B its next value g (*green*) at (3). This assignment at (3) is consistent as the neighbors A and B now have different colors, so we instantiate C next. The assignments at (4) and (5) are not consistent and lead to backtracking. At (6) we find a consistent assignment $(A = r, B = g, C = b)$ from which we can traverse further.

9 Kumar, "Algorithms for Constraint Satisfaction Problems: A Survey".
10 Russell and Norvig, *Artificial Intelligence: A Modern Approach*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Solving a CSP with finite domains is in general a NP-complete problem. Given a CSP with *n* variables and the maximum domain size d of any variable the number of possible complete assignments is $O(d^n)$. This is exponential in the number of variables. Therefore, in the worst case, a CSP can only be solved in exponential time. Backtracking is a first step to reduce the runtime of solving a CSP. Many other methods exist that can speed up the solving process, two prominent are *constraint propagation* and *heuristics for variable and value ordering*. Choosing wrong heuristics and propagation techniques for a given problem can increase the search time, if they add unnecessary overhead compared to a search without using them.[11]

Constraints propagation is used to reduce the search space by deleting elements from domains. A simple constraint propagation method is *forward checking*. Whenever we assign a variable X_i a value we look at the domains for all variables X_j that share a constraint with X_i. We delete every value from the domain of the variables X_j that are inconsistent with the value we choose for X_i. Let's assume that we assign our variable A from the map-coloring problem the color *red* . The neighbors of A (B and C) cannot be assigned the color *red* anymore, as this would violate the constraint that no adjacent territories can have the same color. Therefore we can delete *red* from the domains of B and C. In conclusion, when instantiating B or C we have smaller domains to consider which reduces the width of our search tree and ultimately the number of assignments we have to check. Other, even stronger constraint propagation methods are for example *arc consistency* and *k-consistency*.[12]

In the map-coloring example we used strict sequences for picking a variable to assign next and which value in the domain to assign next. We instantiated the variables from A to F and always tried *red*, then *green*, then *blue*. We can use heuristics to dynamically decide which variable to pick up next and which value to assign next. This can lead to violating constraints faster which eliminates bigger parts of the search tree. Examples for such heuristics are:[13]

- **Degree heuristic:** Choose the variable with the most constraints on remaining variables.
- **Minimum remaining values:** Choose the variable with the fewest possible values.
- **Least constraining value:** Choose the value that rules out the fewest values in the remaining variables.

Other variants of CSP exist that extend or change the definition we give in Definition 3.1. These extensions or changes alter the possible solving strategies of a CSP. Although they are out of scope for this thesis, two examples are (taken from Russell and Norvig[14]):

- **Variables with infinite domains:** For example, when scheduling construction jobs, each job is represented by a start date. The start dates could be the number of days with respect to the current date. This creates domains of integer values from zero to infinity.
- **Preference constraints:** These constraints indicate which solutions are preferred and can be violated at a cost. For example, in an university timetabling problem, Professor A prefers to teach in the morning whereas Professor B prefers to teach in the evening. The preference can be encoded as cost on assigning individual variables in respect to an overall objective function. Assigning a non preferred value to a variable costs more against the overall objective function.

11 Kumar, "Algorithms for Constraint Satisfaction Problems: A Survey".
12 Ibid.
13 Russell and Norvig, *Artificial Intelligence: A Modern Approach*.
14 Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.3 Subgraph isomorphism

Finding circuits that are subcircuits of a target circuit and match to a given pattern constitutes a *subgraph isomorphism problem*. This search is also called graph-based pattern matching and has appliances in many areas such as biology, computer vision, computer aided design, and intelligence analysis..[15] In the following, we will use the definition given in Definition 3.2 when we refer to the term *graph*, Definition 3.3 when we refer to the term *subgraph isomorphism*, and Definition 3.4 when we refer to the term *subgraph isomorphism problem*. Although neither our pattern graphs modeled in PGSL nor the design-graph abstraction use edge labeling, we include this form of labeling in the definitions for the sake of completeness.

Definition 3.2 (Graph[16]):
A graph G is a triple G = (V, E, L) where

1. *V is a set of vertices,*
2. *E is a set of edges,*
3. *each $e \in E$ is a pair (u, v) where $u, v \in V$, and*
4. *L is a labeling function which maps a vertex or an edge to a label*

Definition 3.3 (Subgraph isomorphism[17]):
Given a pattern graph G_1 and a target graph G_2 a subgraph isomorphism ("embedding") is an injective function $M : V_1 \rightarrow V_2$ that:

1. *preserves adjacency: $\forall u, v \in V_1 : (u, v) \in E_1 \Rightarrow (M(u), M(v)) \in E_2$,*
2. *preserves labeling of vertices: $\forall u \in V_1 \Rightarrow L_1(u) = L_2(M(u))$, and*
3. *preserves labeling of edges: $\forall (u, v) \in E_1 \Rightarrow L_1(u, v) = L_2(M(u), M(v))$*

Definition 3.4 (Subgraph isomorphism problem[18]):
Given a pattern graph G_1 and a target graph G_2, the subgraph isomorphism problem is to find all distinct embeddings of G_1 in G_2.

The subgraph isomorphism problem is a is NP-complete problem, therefore the worst case is that an algorithm solves the problem exponentially in the size of the input graphs.[19] During our research we find three approaches to solve the subgraph isomorphism problem:

1. **Dedicated algorithms based on tree search:** e.g., Ullmann[20] and *VF2* by Cordella et al.[21]

[15] B. Gallagher. “Matching structure and semantics: A survey on graph-based pattern matching”. In: *AAAI Fall Symposium - Technical Report* 6 (Jan. 2006).
[16] J. Lee et al. “An in-depth comparison of subgraph isomorphism algorithms in graph databases”. In: *Proceedings of the VLDB Endowment* 6 (Dec. 2012), pp. 133–144. DOI: 10.14778/2535568.2448946.
[17] Ibid.
[18] Ibid.
[19] Gallagher, “Matching structure and semantics: A survey on graph-based pattern matching”.
[20] J. R. Ullmann. “An Algorithm for Subgraph Isomorphism”. In: *J. ACM* 23.1 (Jan. 1976), pp. 31–42. ISSN: 0004-5411. DOI: 10.1145/321921.321925.
[21] L. Cordella et al. “A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 26 (Nov. 2004), pp. 1367–1372. DOI: 10.1109/TPAMI.2004.75.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

2. **Formulating the problem as CSP**: e.g., Larrosa and Valiente,[22] Solnon,[23] and Zampelli et al.[24]
3. **Using graph partition and labeling:** e.g., *SubGemini* by Ohlrich et al.[25]

We will discuss the approaches with examples in the following paragraphs. The algorithm of Ullmann[26] and the *VF2* algorithm[27] are still widely used as basis for subgraph isomorphism algorithms, but the trend is to move towards CSP approaches as they can offer better performance for large graphs and offer more flexibility to formulate additional constraints on the matching procedure.[28] Last but not least, we also review the algorithm *SubGemini*, as it is an interesting approach using partitioning and incremental labeling.

One of the first and highly-cited approaches concerning the subgraph isomorphism problem is the algorithm proposed by Ullmann in 1976.[29] Although it was designed for undirected, unlabeled graphs, it can be adapted to directed graphs with labels. It is a brute-force tree-search enumeration procedure that prunes branches of the search tree with a filter and refinement step. For a given pattern graph with *m* vertices and a given target graph with *n* vertices, the algorithm uses an mxn binary compatibility-matrix M to find subgraph isomorphisms. A *one* in the *i*-th row and *j*-th row indicates that the *j*-th vertex of the target graph is a candidate for the *i*-th vertex in the search graph. The algorithm starts with an initial matrix M^0 where each row is filled with all candidates for the vertex in the search graph, based on a initial compatibility condition. In their original work this initial compatibility of vertices is based on the notion that the degree of a candidate must be at least be degree of vertex in search graph, but this can easily be extended to matching labels. Subsequently, matrices M are created from the matrix M^0 in a step-by-step fashion. Each new matrix M is a node in the search tree. At every newly created matrix one candidate to search a vertex pair (m, n) is chosen as *compatible pair* and all other elements in row m and column n are set to zero (this is the filtering step). Next, the matrix is refined by checking all ones in the matrix if they can be changed to zero: a pair of pattern and target vertices is only compatible if a compatible pair exists in their neighborhoods. If no subgraph is possible with this matrix (a row is all zeroes), the algorithm backtracks. Overall, the processing time of Ullmann's algorithm is exponential with the size of the graphs and memory intensive as for every step a mxn matrix is created. Therefore it is very expensive for large target graphs.

The *VF2* algorithm[30] is similar to Ullmann's algorithm, but rather than "pruning down" it incrementally "builds up" by extending partial assignments which the authors call states. This leads to reduced memory consumption, as backtracking can be done without the need of storing copies of previous states. VF2 starts with a first vertex in the pattern graph and subsequently one by one extends the matching to a vertices that are connected to the already matched query vertices. At each step a set of heuristics is used to prune down the candidates for extending the partial mapping with an assignment to a vertex. Possible mappings are considered and extended with backtracking. The heuristics are based on analysis of nodes adjacent to the ones already considered in the partial mapping.

A contrast to dedicated subgraph isomorphism algorithms is the formulation of the problem as CSP. The nodes of

[22] J. Larrosa and G. Valiente. "Constraint Satisfaction Algorithms for Graph Pattern Matching." In: *Mathematical Structures in Computer Science* 12 (Aug. 2002), pp. 403–422. DOI: 10.1017/S0960129501003577.
[23] C. Solnon. "AllDifferent-based filtering for subgraph isomorphism". In: *Artificial Intelligence* 174.12 (2010), pp. 850–864. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2010.05.002.
[24] S. Zampelli et al. "Solving Subgraph Isomorphism Problems with Constraint Programming". In: *Constraints* 15.3 (July 2010), pp. 327–353. ISSN: 1383-7133. DOI: 10.1007/s10601-009-9074-3.
[25] M. Ohlrich et al. "SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm". In: *30th ACM/IEEE Design Automation Conference*. June 1993, pp. 31–37. DOI: 10.1145/157485.164556.
[26] Ullmann, "An Algorithm for Subgraph Isomorphism".
[27] Cordella et al., "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs".
[28] C. McCreesh et al. "When Subgraph Isomorphism is Really Hard, and Why This Matters for Graph Databases". In: *J. Artif. Intell. Res.* 61 (2018), pp. 723–759.
[29] Ullmann, "An Algorithm for Subgraph Isomorphism".
[30] Cordella et al., "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

a pattern graph become variables, their candidates domains, and the edges become adjacency constraints of a CSP. Additionally, to ensure that each vertex in a pattern graph is matched to a distinct vertex in the target graph, usually an "all different" constraint is added to the adjacency constraints. With this constraint a CSP solver checks after each assignment extension if the new assignment does not contain a vertex of the target graph more than once. Solving the graph isomorphism problem as CSP is done by backtracking combined with intelligent constraint propagation, variable ordering, and value ordering. Making subgraph isomorphism solving via CSP competitive with the dedicated algorithms boils down to choosing a good mix of constraint propagation (filtering) and variable and value ordering. Finding this good mix is still a topic of research. For example, Larrosa and Valiente[31] introduce a *neighborhood constraint* which represents the notion that a pattern vertex v_i can only be mapped to a vertex v_a in the target graph if all vertices in the neighborhood of v_i can be mapped to vertices in the neighborhood of v_a. Using this constraint, the domain of the variable i that represents v_i is pruned before extending the assignment. Solnon[32] uses a minimum remaining values heuristic picking the variable next with fewest remaining values in its domain. Zampelli et al.[33] even uses a dedicated labeling algorithm to compute labels for vertices in the target graph to remove from domains of the CSP vertices. The labeling is based on vertex degree and is a iterative procedure.

Another noteworthy algorithm is SubGemini.[34] Their method finds a given circuit as subcircuit in a target circuit using a labeling algorithm and breadth first search (BFS). Their approach has two phases. Phase one identifies all possible locations of the circuit in the target circuit. First, they identify a key vertex in the circuit that they intend to search for. In a second step vertices in the target circuit are identified which match the key vertex. In the second phase, starting from the key vertex candidate, in multiple iterations, they label the surrounding vertices in both subcircuit and target circuit. If all labels from the search circuit are found in the target circuit a subcircuit match is found.

3.4 Regular expressions

Regular expressions, also called *regexes*, are a powerful tool to match string patterns. The syntax of the pattern graph specification language (PGSL), and especially its quantifiers, is influenced by regular expressions. The concept of regular expressions arose in the field of theoretical computer science, specifically in the subfields of automata theory. Kleene[35] was one of the first to develop regular expressions to describe *regular languages* in a mathematical notation . One of the first uses of regular expressions in a program was done by Ken Thompson when he built a regex notation into the editor *QED* to match patterns in string files. Regexes as early form were first standardized in the *POSIX standard* in 1986. Since then, many different styles of regular expressions have been developed and regular expressions are nowadays widely supported in programming languages, text processing tools, advanced text editors (e.g., my editor of choice *vim*), and many other tools in the field of computer science. In the context of programming languages the implementation of regex support is often called *regex engine*.[36] As it is one of the most widely adapted standards, we adhere to the IEEE POSIX ERE (Extended regular expression) standard[37] for formulating regular expressions.

[31] Larrosa and Valiente, "Constraint Satisfaction Algorithms for Graph Pattern Matching."
[32] Solnon, "AllDifferent-based filtering for subgraph isomorphism".
[33] Zampelli et al., "Solving Subgraph Isomorphism Problems with Constraint Programming".
[34] Ohlrich et al., "SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm".
[35] S. Kleene. *Representation of Events in Nerve Nets and Finite Automata.* Memorandum (Rand Corporation). Rand Corporation, 1951.
[36] J. Friedl. *Mastering Regular Expressions.* Aug. 2006. ISBN: 0596528124.
[37] "IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))". in: *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)* (Dec. 2008), pp. 183–195.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 3.1: Selected metacharacters used with IEEE POSIX ERE.

Metacharacter	Name	Description
.	*dot*	Matches any single character
[ ]	*character class*	Matches *any character contained in the brackets*
[^]	*negated character class*	Matches *any character not contained in the brackets*
?	*question*	Matches the preceding element *zero or one time*
*	*star, "Kleene star"*	Matches the preceding element *zero or one times*
+	*plus, "Kleene plus"*	Matches the preceding element *one or one times*
{*m*,*n*}	*specified range*	Matches the preceding element *at least m and at most m times*
|	*or, bar*	Matches either the *element left or right of the bar*
( )	*parenthesis*	Defines the element inside the parenthesis as a *marked subexpression*
\1, \2, ...	*backreference*	Matches the previously matched first, second, etc. element enclosed in parenthesis

Regular expressions can be viewed as a language with *text* (a sequence of characters) as words and *metacharacters* (characters with special meaning) as grammar. Combining metacharacters and text with a specified set of rules creates a regular expression. A regular expression is build up from small building blocks. Combined, these small building blocks form powerful string matching patterns. A selection of metacharacters, their names, and a short description is shown in Table 3.1. Regular-expression matching can be implemented as finite state machines (FSM), in particular with deterministic finite automata (DFA) and nondeterministic finite automata (NFA).[38] We consciously avoid elaboration how regexes can be converted to FSMs, and rather provide regex examples that allow a reader to grasp the concept of regular expressions.

The simplest regular expression is one or more characters, as for example seen in Equation (3.1).

$$\text{calendar} \tag{3.1}$$

This regular expression (Equation (3.1)) matches all occurrences of the word "calendar" in a given text. If we want in addition to match possible misspellings of this word, we can extend our regex with a *character class* [ae] that matches to a character in the set $\{a, e\}$:

$$\texttt{c[ae]l[ae]nd[ae]r} \tag{3.2}$$

For instance, calandar, calender, and celandar are also matched by the regular expression given in Equation (3.2). If we want to match from a set of regular expressions, we can use the *bar* metacharacter. For example the regular expression in Equation (3.3) matches to the word foo or bar in a given text. Such alterations are often grouped using the *parenthesis* metacharacters.

$$\texttt{(foo|bar)} \tag{3.3}$$

Regular expressions allow matching of a subexpression multiple times in sequence by quantification. Quantification is done by adding one of the quantification metacharacters after the subexpression. An example is searching for hexadecimal numbers. One hexadecimal digit can be matched by a character class [a-f0-9]. For example, if we want to find *one or more* hexadecimal digits as sequence we use the + quantifier (see Equation (3.4a). If we want to match an interval of digits we can use a ranged quantification from minimal *n* to maximal m (see Equation (3.4b)). Matching the regular

[38] A. Brüggemann. “Regular Expressions into Finite Automata.” In: *Theor. Comput. Sci.* 120 (Jan. 1993), pp. 197–213.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

expression for hexadecimal digits exactly n times is also possible (see Equation (3.4c)).

$$[\text{a-f0-9}]+ \quad (3.4a)$$

$$[\text{a-f0-9}]\{1,8\} \quad (3.4b)$$

$$[\text{a-f0-9}]\{8\} \quad (3.4c)$$

As a final example we provide a real-world problem: the verification of email addresses. An email address consists of a *local-part* followed by an '@' and a *domain-part*. Local-part and domain-part each have a set of allowed characters. With a regular expression built from three subexpressions (see Equation (3.5)) we can match valid email addresses.

$$[\text{A-Z0-9+_.-}]+@[\text{A-Z0-9.-}]+ \quad (3.5)$$

An example for a valid address that is matched by this regular expression is martin@mosbeck.at. A invalid email address, for example martin.mosbeck.at, is not matched, as a valid email address needs to have an '@' sign that separates the local-part (martin) and the domain-part (mosbeck.at).

3.5 Regular graph expressions

PGSL uses the concept of regular expressions to quantify the occurence of subpatterns. Regular expressions are used in conjunction with graph matching in other fields, in particular querying graph-structured data such as found in social networks, dynamic network traffic, biology and intelligence analysis. In these graph structures nodes are attributed with labels that represent properties, and attributes and edges that are labelled with relations between the two nodes that share an edge. Two types of queries are used:[39] (1) reachability queries, and (2) graph pattern queries. Reachability queries are used to find out whether a path exists between one node with a specific label and another node with a specific label. A valid result path can be constrained by the number of nodes that are on the path, the labels of the nodes on the path or the labels of edges on the path. Graph patterns are built using reachability queries. Graph pattern queries are similar to patterns we are searching for in this thesis. Graph pattern queries are searches for subgraphs of the data graphs where the subgraph has to be isomorphic to a pattern graph.

However, quantifications using PGSL are different from using reachability queries. Quantifications quantify the occurrence of subpatterns and every element of a serial chain has to have the same topology and set of node types. In addition, we do not care about edge labels and can also quantify the occurrence of subpatterns in parallel.

Besides subgraph isomorphism, the concept of graph simulation (a relaxed version of subgraph isomorphism) is sometimes used to find matches for graph pattern queries (e.g., Ma et al.,[40] Fan et al.[41]). With graph simulation for example edges of a pattern graph can be "mapped to (bounded) paths instead of edge-to-edge mappings".[42] A comparison of

[39] W. Fan et al. "Adding regular expressions to graph reachability and pattern queries". In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pp. 39–50.
[40] S. Ma et al. "Strong Simulation: Capturing Topology in Graph Pattern Matching". In: *ACM Transactions on Database Systems (TODS)* 39 (Jan. 2014), p. 4. DOI: 10.1145/2528937.
[41] Fan et al., "Adding regular expressions to graph reachability and pattern queries".
[42] Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

subgraph isomorphism and graph simulation can be found in the work of Seba et al..[43] To empathize the concepts of reachability queries and graph patterns and their relation we present two example variants of reachability queries and graph pattern queries.

Barceló et al.[44] describe *regular path queries* (reachability query) as a search for a pair of nodes connected by a path with the constraint that all the node labels on the path have to match a given regular expression. In the context of the analysis of the authors a *graph pattern* is a graph database with constant nodes, node variables, and edges that are specified with regular path queries. Node variables are filled during the match process.

Fan et al.[45] also use reachability queries. In addition, they add search conditions to nodes, in particular what labels are allowed for a node to match a node in the pattern. Their path matching is via regular expressions, but rather than defining which node labels are allowed to appear on the path they specify which edge labels are allowed. Additionally, they can limit the number of hops (number of intermediate nodes) on a path. Built on these reachability queries, a pattern graph is a set of nodes with reachability queries that connect these nodes. As example to use their pattern queries the authors generated and used a "terrorist organization collaboration network, from 81800 worldwide terrorist attack events, where each node represents a terrorist organization (TOs) with attributes such as name (gn), country, target type (tt), and attack type (at); and edges bear relationships, e.g., international (resp. domestic) collaborations ic (resp. dc), from organizations to the ones they assisted or collabo- rated in the same country (resp. different countries)."[46]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

43 H. Seba et al. “Comparison Issues in Large Graphs: State of the Art and Future Directions”. In: *ArXiv* abs/1502.07576 (2015).

44 P. Barceló et al. “Querying Regular Graph Patterns”. In: *J. ACM* 61.1 (Jan. 2014). ISSN: 0004-5411. DOI: 10.1145/2559905. URL: https://doi.org/10.1145/2559905.

45 Fan et al., “Adding regular expressions to graph reachability and pattern queries”.

46 Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Chapter 4

Search pattern modeling

We use pattern graph specification language (PGSL) to model pattern graphs that represent the structural kernels of functional primitives (e.g., counters, state machines, ...) which we want to find in a hardware description language (HDL) design. PGSL was envisioned by Krieg[1] and fine-tuned in this thesis. A structural kernel is the graph structure that describes the core functionality of a functional primitive. For example, the core functionality of a counter is a register whose value is incremented. This core functionality is represented by a structural kernel in a synthesized design that we aim to find with our search. For the counter this structure is a graph containing a register cell, an addition cell, and a feedback-chain containing multiplexer cells. In PGSL, we connect pattern cells of specified types with operators to form a graph. In the context of PGSL patterns, we use the terms *node* and *cell* synonymously, as the nodes of our pattern graph match cells in the design graph.

The instances of a functional primitive can vary structurally. For example, different counters can have feedback chains with different lengths. The feedback chain implements the control path of the counter and can be of variable length as different instances of counters are influenced by different control signals (e.g., *reset*, *preset*). To model structural variability, any PGSL subpattern, which itself is a subgraph of the pattern graph, can be quantified. With quantifications we can quantify the occurrence of subpatterns (for example *none or more* or *maximum M times*). Putting it all together, a simple pattern for a counter in PGSL is:

```
[[ $add -> $mux >* -> $dff ]] ;
```
(4.1)

This pattern gives a good first impression of PGSL. It uses

(1) node types $add, $dff, and $mux which we want to match,
(2) operators ->, >, [[, and]] to define the structure of the graph,
(3) a quantifier * to quantify the occurrence of the $mux subpattern, and
(4) the pattern termination character ‘;’.

[1] C. Krieg. “Pattern-Based Hardware Trojan Characterization for Design Security Assessment”. PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

```
⟨pattern⟩ ::= ⟨opt_macro_definition_list⟩ ⟨opt_p_label⟩ ⟨expression_list⟩ ';'

⟨macro_key⟩ ::= '@' ⟨STRING⟩

⟨macro_definition⟩ ::= ⟨macro_key⟩ '=' ⟨node_type_spec⟩ ';'

⟨macro_definition_list⟩ ::= ⟨macro_definition⟩
    | ⟨macro_definition_list⟩ ⟨macro_definition⟩

⟨opt_macro_definition_list⟩ ::= ⟨empty⟩ | ⟨macro_definition_list⟩

⟨expression_list⟩ ::= ⟨expression⟩
    | ⟨expression_list⟩ ',' ⟨expression⟩

⟨expression⟩ ::= ⟨node_type_spec⟩ ⟨opt_quantification⟩ ⟨opt_e_label⟩
    | ⟨expression⟩ '->' ⟨expression⟩
    | ⟨expression⟩ '||' ⟨expression⟩
    | '(' ⟨expression⟩ ')' ⟨opt_quantification⟩ ⟨opt_e_label⟩
    | '[[' ⟨expression⟩ ']]' ⟨opt_e_label⟩
    | ⟨e_label⟩

⟨inner_connector⟩ ::= '>' | '|'

⟨quantifier⟩ ::= '#' | '~' | '+' | '*'

⟨opt_quantification⟩ ::= ⟨empty⟩ | ⟨quantification⟩

⟨quantification⟩ ::= ⟨inner_connector⟩ ⟨quantifier⟩
    | ⟨inner_connector⟩ '{' ⟨UNSIGNED⟩ '}'
    | ⟨inner_connector⟩ '{' ⟨UNSIGNED⟩ ',' '}'
    | ⟨inner_connector⟩ '{' ⟨UNSIGNED⟩ ',' ⟨UNSIGNED⟩ '}'

⟨node_type_spec⟩ ::= ⟨node_identifier⟩
    | ⟨node_type_spec⟩ '|' ⟨node_identifier⟩
    | ⟨macro_key⟩

⟨node_identifier⟩ ::= '$' ⟨STRING⟩ | '\' ⟨STRING⟩

⟨opt_e_label⟩ ::= ⟨empty⟩ | ⟨e_label⟩

⟨e_label⟩ ::= ':' ⟨STRING⟩

⟨opt_p_label⟩ ::= ⟨empty⟩ | ⟨p_label⟩

⟨p_label⟩ ::= ⟨STRING⟩ '='

⟨STRING⟩ ::= [a-zA-Z_0-9]+

⟨UNSIGNED⟩ ::= [0-9]+
```

Figure 4.1: The grammar of the pattern graph specification language (PGSL) in backus–Naur form (BNF)

In the following sections we describe the elements of PGSL and how we use PGSL to model patterns for the functional primitives counter, state machine, and elements of encoders/decoders. As an overview and reference we list the full grammar of PGSL in backus–Naur form (BNF) in Figure 4.1.

4.1 Structural elements of PGSL

Node-type specification and macros

The basis of each PGSL pattern are node types we want to match. As we search in the target design graph representation of designs (see Section 3.1) we use the node types that are present in the design graphs. These are the cell types from the *Yosys* RTL intermediate language (RTLIL)[2] (leading $; e.g., $dff) and additional node types to match constants (\CO),

[2] C. Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

(\CO -> $eq) |+ -> @MUX

Figure 4.2: Illustrating the block structure of a PGSL pattern. Block (1), (2), and (4) are single nodes, block (3) is a composite node containing block (1) and (2). (0) is the whole pattern, that itself is an unquantified composite node.

primary inputs of the design (\PI), and primary outputs of the design (\PO). A node-type specification on its own is already a valid pattern. For example the following pattern describes that we want to find all data flip-flop $dff nodes in a design:

$$\$dff \;\; ; \tag{4.2}$$

As structures with similar functionality can be implemented in RTL with different cell types, PGSL allows node specifications that represent a match in a set of cell types. These sets can be specified by using the *logical or operator* '|'. For example, to specify a node in our pattern that matches different types of register cells, we can write:

$$\$dlatch \;|\; \$adff \;|\; \$dff \;\; ; \tag{4.3}$$

Patterns tend to become harder to read with these node-type-set specifications. Therefore, PGSL offers the possibility to define macros for a group of types for later use in patterns. If we want to define a macro *REG* (for "registers") for different types of registers, we can write:

$$@REG = \$dlatch \,|\; \$adff \;|\; \$dff \;\; ; \tag{4.4}$$

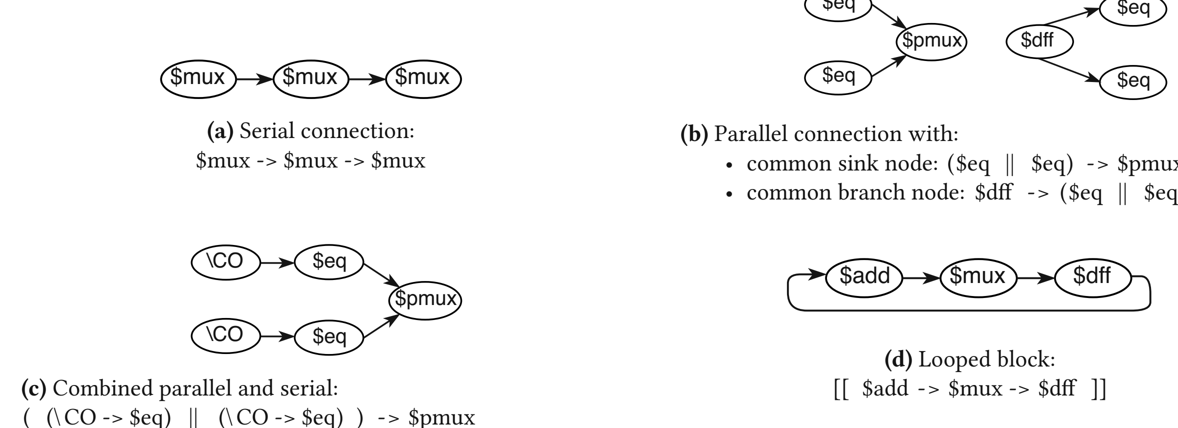
Subsequently, we use @REG in our patterns to indicate that we want to use the node-type group.

Blocks

A PGSL pattern consists of *blocks* that are connected via *structural operators*. Blocks are surrounded by parentheses '(' and ')'. We call the simplest block a *single node* as it only contains one node-type specification and does not need to be surrounded by parentheses. Combining single nodes with structural operators and surrounding them by parentheses forms a *composite node* block. Each block is a subpattern of the whole PGSL pattern and can be quantified. Composite nodes themselves can both contain single nodes and composite nodes, therefore a PGSL pattern is hierarchical. Finally, a PGSL pattern as a whole is an unquantified composite node. An illustrative example of this hierarchical block structure of a PGSL pattern is shown in Figure 4.2. The pattern as a whole forms a composite node (0). It contains the subpatterns (3) and (4). The subpattern (3) is a composite node with two subpatterns, the single nodes (1) and (2). The subpattern (4) is a single node. We attribute a block with a set of left nodes and right nodes. These are the nodes which are the left and right boundary nodes when reading the block pattern from left to right and viewing the resulting pattern as a graph. Single nodes only have one left node that is also the right node as there is only one node contained in the block. Krieg[3] calls single nodes "trivial nodes" and composite nodes just "blocks". This difference in nomenclature is intentional, as these names are also used in the implementation of the search method presented in this thesis.

[3] Krieg, "Pattern-Based Hardware Trojan Characterization for Design Security Assessment".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Serial connection:
$mux -> $mux -> $mux

(b) Parallel connection with:
- common sink node: ($eq || $eq) -> $pmux
- common branch node: $dff -> ($eq || $eq)

(c) Combined parallel and serial:
((\CO -> $eq) || (\CO -> $eq)) -> $pmux

(d) Looped block:
[[$add -> $mux -> $dff]]

Figure 4.3: Visualizing the use of *outer operators* in PGSL patterns and there equivalent graphs.

Structural operators

PGSL uses two variants of structural operators: *inner operators* and *outer operators*. Outer operators specify the connectivity between blocks. Blocks can be serially connected (operator ‘->’) or specified to be in parallel (operator ‘ || ’). A serial connection between a block *(1)* to block *(2)* (*(1)* -> *(2)*) always connects the right nodes of block *(1)* to the left nodes of block *(2)*. We established this restriction as digital designs show an input-to-output and output-to-input connectivity characteristic. The parallel operator specifies that two blocks are in parallel. To form a valid PGSL pattern, parallel blocks need a common connection to a node (either as sink or branch or both). Finally, blocks can be encapsulated in a loop with *loop begin* (operator ‘[[’) and *loop end* (operator ‘]]’). This connects all right nodes of a block to all left nodes of the block. A visualization of example patterns with *outer operators* is shown in Figure 4.3. *Inner operators* specify the connectivity inside of a quantified block, we discuss them in the next section.

Quantification

Quantification is the PGSL feature to model structural variability of patterns. Quantifications have always two elements: the *inner operator* and the *quantifier*. Quantifications operate on the preceding block. This leads to a common syntax for quantification:

$$\text{block inner-operator quantifier} \tag{4.5}$$

Inner operators come in two variants: (1) serial quantification (inner operator ‘>’), and (2) parallel quantification (inner operator ‘|’). The syntax of quantifiers is inspired by regular expressions and specifies the number of occurrences we want to match. A full list of all quantifiers available in PGSL can be seen in Table 4.1

The serial-quantification inner-operator specifies that we want to match connected chains of the block that is quantified. An example where we use this feature is variable-length multiplexer chains that are feedback paths of counters and state machines. Instead of specifying multiple patterns for different counter instances with different lengths of multiplexer chains, we can write:

$$\text{\$mux >*} \tag{4.6}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Table 4.1: Available quantifiers in PGSL and their meaning. Quantifiers specify the number of occurrences of a quantified block to match.

Quantifier	Meaning
+	Match one or more times
*	Match zero or more times
#	Match even number of times
~	Match odd number of times
{N}	Match exactly N times
{N,}	Match at least N times
{N,M}	Match from N to M times

The pattern in Equation (4.6) specifies that we want to match *zero or more* multiplexers connected in serial. The "zero occurrence" case is special for serial quantifications. In this case we interpret the block as shorted and replace the block with a connection from it's block on the left to its block on the right. We call this a "possibly shorted" node.

The parallel-quantification inner operator specifies that we want to match occurrences of the quantified block in parallel. Similar to blocks with a parallel outer operator, parallel-quantified blocks need an adjacent anchor point as either a connection to a sink node that cannot be parallely quantified or from a branch node that cannot be parallely quantified, or both. An example, how we use parallel quantification, are parallel comparison branches that connect to a common multiplexer as sink node:

(\CO -> $eq) |+ -> @MUX ; (4.7)

In this case (Equation (4.7)), the quantification '|+' specifies that we want to match *one or more* instances of the subpattern (\CO -> $eq) in parallel. Similar to the serial quantification, parallel-quantification containing zero occurrence is a special case: the node can be replaced by a split in the graph, therefore we call parallel-quantified blocks with possible zero-occurrence "possibly split" nodes.

Expressions, labels, and references

It is possible to specify PGSL patterns as a list of expressions, that together form the pattern. The expression list is separated by commas. To connect blocks across the expressions, we can label blocks (':' followed by a string) and use the labels as references. An easy example how to use labels and references, is rewriting the pattern shown in Figure 4.3d without using the loop operator:

$add:adder -> $mux -> $dff : register , : register -> :adder ; (4.8)

The second expression (: register -> :adder) connects the right side of the data-flip-flop node that is labeled to the left side of adder node.

4.2 Creating PGSL patterns

A verification- or reverse engineer who wants to identify functional primitives in a design, for example counters and state machines, has to model a suitable PGSL pattern. The first step is to identify the functional kernel of the element he or she wants to match with the pattern. Functional kernels manifest as structural kernels in design graphs. To

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

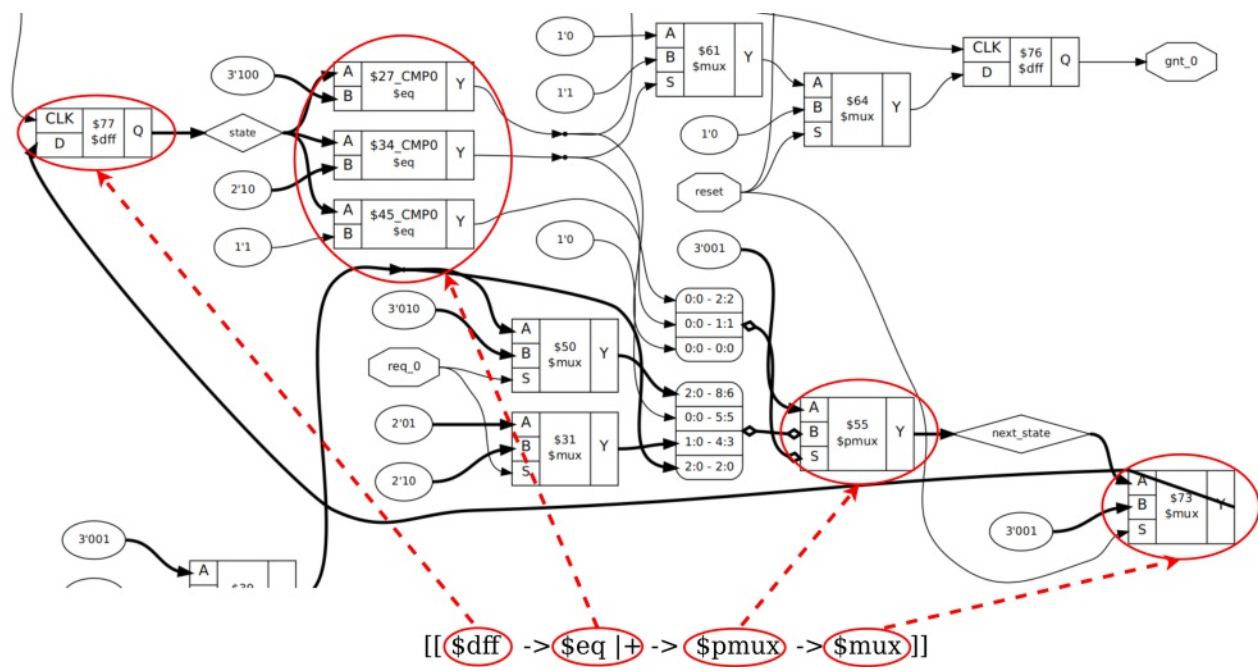


Figure 4.4: *Yosys* RTLIL graph of a simple arbiter state machine with three states (encoded as "1", "10", "100"). The cells we choose as part of the structural kernel are marked. We use this structural kernel as first version for a PGSL pattern for state machines.

determine the structural kernel, we can either use an experience-based trial and error approach or examine instances of the functional primitive to discover the common structure. The second variant involves taking designs that contain the specific functional primitive and analyzing their resulting representation in the synthesized design. In this step we use RTL netlists, as they contain ports and signal names, which allows us to trace primitives from their HDL specification. Translating the obtained knowledge to the design graph abstraction is an easy step, as we just have to eliminate cell ports and wires as the design graph abstraction does. Once we obtained a common structure, we model a first version of the PGSL pattern. By using the pattern in searches on multiple designs, we find cases where our pattern fails, and subsequently fine-tune the pattern. For now, the process of pattern creation is a manual task, needing a human to model, try, refine and try again to find the most generalized version of a pattern for a target functional primitive. In the following subsections we elaborate the creation of the patterns for state machines, counters and encode/decode elements, which we use for our experimental examination of the search method presented in this thesis (see Chapter 6).

State machines

To create a pattern for state machines, we first analyze the core functionality of state machines. A state machine contains a state register with the current state value. Depending on the current state and values of control signals, a next state value is calculated and assigned as current state. With this in mind, we next analyze the RTL netlist of a simple (few control signals) state-machine with three states as shown in Figure 4.4. Analyzing the structure allows us to model a first version of a PGSL pattern. We see that the signal *state* (in a diamond shape) is attached to a data flip-flop cell ($dff). This *state* is connected to comparator cells ($eq). The results of the comparisons are fed to a parallel multiplexer cell ($pmux), whose output is the signal *next_state* (also in diamond shape). The *next_state* is fed back via a multiplexer cell ($mux) to the state register.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

With this knowledge, we can model subpatterns and connect them to a PGSL pattern. As the comparator cells are in parallel, and we expect state machines to have variable amounts of states, we can model a quantified block:

`$eq |+` (4.9)

Next we can connect the other nodes ($dff , $pmux, $mux) we choose as part of the structural kernel as seen in Equation (4.10).

`$dff -> $eq |+ -> $pmux -> $mux` (4.10)

Finally we add the feedback as loop operator to the pattern and obtain the first version of our pattern for state machines:

`[[ $dff -> $eq |+ -> $pmux -> $mux ]] ;` (4.11)

By using the pattern to search in bigger and more complex designs, we find improvements to our pattern and incrementally improve the pattern:

- The state register can also be of the types $adff and $dlatch, therefore we created a macro to cover the different types: @REG = $dlatch | $dff | $adff
- The feedback path can be a chain of *zero or more* multiplexers of the types $mux and $pmux. Therefore we first created a macro @MUX = $mux | $pmux and quantified the block as chain with the zero or more occurence quantifier '*': @MUX >*
- *Yosys* optimizes comparisons to the state value logical "0" as $logic_not cells and comparators can also take the form of multiplexer cells $mux. As these two cases do not appear in all cases, we grouped them, quantified them in parallel with the zero or more quantifier and use the parallel outer-operator to signify that they are parallel to the $eq cells, leading us to: ($mux | $logic_not) |* || $eq |+

Putting it all together we obtain the final pattern for state machines:

`[[ @REG -> ( ( $mux | $logic_not ) |* || $eq |+ ) -> $pmux -> @MUX >*]] ;` (4.12)

Counters

To create a PGSL pattern for counters we reuse the reasoning that lead us to Equation (4.1). With the knowledge, that feedback paths can consist of different multiplexer types and registers are of the type-group @REG we can model a pattern:

`[[ $add -> @MUX >*-> @REG ]] ;` (4.13)

During testing of this pattern, we came up with a second pattern, that matches only counters that are incremented by a constant value:

`\CO -> [[ $add -> @MUX >*-> @REG ]] ;` (4.14)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Using these two distinct patters, we can easily classify search results for counters into two categories: (1) incremented by constant, or (2) incremented by another source (e.g., the value of a register). Knowing what kind of counters are present in a design leads to a better design understanding.

Encode/decode elements

Another functional primitive of interest in digital designs are encoder and decoder trees as they are for example used to access memory and therefore can help us identify memory controllers. As PGSL does not yet support modelling tree structures (possible future work) we developed two patterns to match to elements of encoders and decoders. It is of interest to find decoders and encoders as we model elements of encoder and decoder trees as parallel comparisons to constants that are connected to a common multiplexer cell. This leads us to the following pattern, which reads as "parallel blocks of constants connected to comparators with a common multiplexer sink node":

```
( \CO -> $eq ) |+ -> @MUX ;
```
(4.15)

During testing of this patterns, we found an additional variant of this pattern. This variant checks if at least one of the comparisons is true. *Yosys* implements this as an added or-reduction operation with a $reduce_or cell:

```
( \CO -> $eq ) |+ -> $reduce_or -> @MUX ;
```
(4.16)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 5

Search methodology and implementation

To find patterns modeled with the pattern graph specification language (PGSL) in a given hardware description language (HDL) design, we implement our search methodology as a *Yosys*[1] plugin. *Yosys* is an open-source synthesis suite and offers a plugin system. Compared to proprietary synthesis suites, *Yosys* offers access to its internal data structures with an application programming interface (API) and allows us to easily insert additional processing stages, in our case the pattern search on synthesized HDL designs. The search operation has two inputs: (1) a design graph, the graph abstraction of the target synthesized HDL design at register transfer level (RTL) (see Section 3.1) level, and (2) a PGSL pattern in textual representation as the model of the structural kernel of a functional primitive which we want to find in the given design. Our search operation can be broken down into following steps:

1. **PGSL lexing and parsing:** We transform the textual PGSL pattern into single nodes and composite nodes that form a hierarchy for our search strategy. These nodes become parts of a search graph which we call *pattern graph*. The pattern graph is more than just an abstract syntax tree (AST). Its nodes and their adjacency guide the overall search strategy. Each node represents a subpattern we match in the design graph. Search results for a subpattern are stored in the respective node. Additionally, in the parsing step, we profile the whole pattern with regard to the types that appear in the pattern. We use that information in a subsequent filtering-step to reduce the overall search complexity.

2. **Creation of a filtered design graph:** We create a filtered design graph that only contains nodes and connections that are relevant for the search operation, i.e., we only keep nodes with types that appear in the PGSL pattern and the connections between these nodes. This initial filtering reduces the search space and therefore reduces search complexity. Additionally, we store the IDs of nodes in the design graph grouped by each node type that appears in the pattern. This way, we only have to iterate once through all nodes of the design graph.

[1] C. Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/.

3. **Hierarchical creation and combination of candidates:** We search for nodes in the design graph that match subpatterns of the pattern graph. We call matches *candidates* and combine candidates to solve for subpatterns higher in the hierarchy. Combining is performed with regard to the candidates and structural constraints posed by the pattern and the candidates of nodes. To check the constraints and create combined candidates, we use a custom constraint satisfaction problem (CSP) solver. We discuss the reasons for implementing a custom CSP solver instead of using an established CSP solver in the subsection that describes the candidate combination process. The hierarchical match-and-combine process terminates if no candidate for a subpattern can be found or the top-level pattern is matched. A candidate for the top-level pattern, which is the whole pattern itself, is a result of the search operation.

4. **Result post-processing and visualization:** We assemble graphs from the top-level candidates as lists of nodes in the design graph and their adjacency matrices. As the search operation can produce duplicates (e.g., due to permutations), we check the graphs for uniqueness and eliminate duplicates if necessary. We save the graphs, which are subgraphs of the design graph, in data structures that are accessible to *Yosys* plugins. Additionally, the graphs are stored in human-readable form and as a search summary, that lists and visualizes the result graphs.

5.1 Parsing

As a first step in the search process, we transform a given PGSL pattern into a structure that encompasses the meaning of the pattern and that can be used in the search process, namely the pattern graph with single nodes and composite nodes. In this context, PGSL is a language with context-free grammar whose rules can be expressed with a backus–Naur form (BNF) (we show the BNF of PGSL in Figure 4.1). To parse the PGSL language we use a lexer-to-parser chain, specifically the lexer-generator *flex*[2] and the parser-generator *bison*.[3] A lexer recognizes lexical patterns in a given language by matching sequences of characters and identifying them as tokens of the language (e.g., '->' is converted to the token *SERIAL*). A parser uses the token sequence of a lexer to check the given text (in our case the PGSL pattern) against the rules specified by the language and creates a data structure that encompasses the meaning of the text.

PGSL patterns model series-parallel graphs[4] which elements are either combined in serial or in parallel. Therefore, the desired structure after parsing is also a graph. During parsing, we identify two types of nodes in a given pattern: (1) single nodes, and (2) composite nodes and assign each a unique pattern-node ID (*pn_id*). We store the set of pattern nodes and their characteristics (quantification, adjacency, contained nodes) with their IDs in a table. Single nodes specify one or a set of node types we want to match a single cell in the target design graph. Composite nodes are created by grouping multiple single nodes or/and composite nodes with the adjacency inner operators serial '->' or parallel ' || ' and enclosing them with the grouping-operators '(' and ')'. We store the connections inside of a composite node as adjacency matrices of the pattern-node IDs that are contained in the composite node. The definition of single nodes and composite nodes can be followed by a quantification which implies that the node is to be seen as a stage that is either to be matched in serial or parallel a number of times by a given interval. To reduce duplication in the quantification-solver code, quantified single nodes are stored in a composite node and the quantification is stored with this node.

[2] *Flex: The Fast Lexical Analyzer.* https://github.com/westes/flex.
[3] *Bison general-purpose parser generator.* https://www.gnu.org/software/bison/.
[4] R. Duffin. "Topology of series-parallel networks". In: *Journal of Mathematical Analysis and Applications* 10.2 (1965), pp. 303–318. ISSN: 0022-247X. DOI: https://doi.org/10.1016/0022-247X(65)90125-3.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

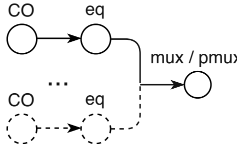


Figure 5.1: Graphical representation of the pattern shown in Equation (5.1).

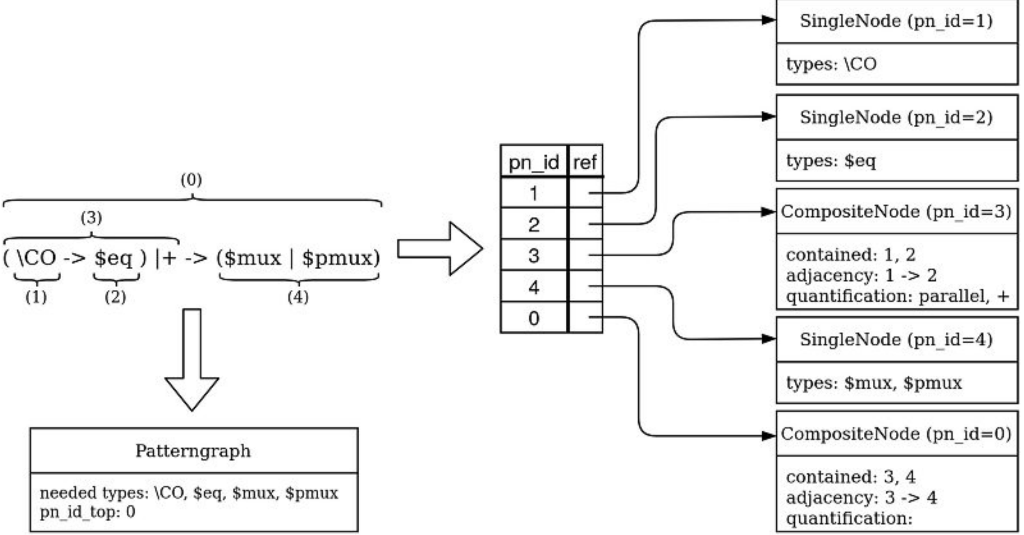


Figure 5.2: Illustrating the result of parsing for an example pattern. The pattern is parsed from left to right, the *pn_id* is incremented starting at 1 (*pn_id=0* is reserved for the top composite node). (1) and (2) in the pattern are parsed as single nodes and combined to a composite node (3), that contains the two nodes and their connection information. (4) is parsed to a single node. The pattern as a whole (0) forms a composite node as top-level. The pattern graph structure stores the top *pn_id* and the contained node types of the pattern.

The top level of a PGSL pattern is a composite node with the *pn_id=0*. Additionally, during parsing of a PGSL pattern we profile which node types appear in the pattern which we call "needed types" of the pattern graph. The result of a parsing process can be illustrated by an example parsing process for the pattern shown in Equation (5.1). A graphical representation for this pattern can be seen in Figure 5.1.

$$(\ \backslash\text{CO -> \$eq} \ ) \ |+ \ \text{->} \ (\text{\$mux} \,|\, \text{\$pmux}) \ ; \tag{5.1}$$

Equation (5.1) is the pattern *Encode/Decode Element 1* we use in our experiments (see Chapter 6), the macro (@MUX) is unpacked as ($mux | $pmux) for the sake of clarity. We illustrate the parse process and result for this specific pattern in Figure 5.2. Parsing creates a pattern-node table that contains unique pattern-node IDs and stores references to all the single nodes and composite nodes and a pattern graph that contains the top pattern-node ID and the list of "needed types". Each of the single nodes and composite nodes also track which of the contained nodes are on the left and right side of the node. This information is needed to properly resolve the nodes to candidates (i.e, subgraphs of the target design graph) during the matching process. For example, the left side of composite node with *pn_id=3* is the single node with *pn_id=1*. We use the pattern graph and pattern-node table in the subsequent phases of the search operation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Table 5.1: Average decrease of the search space parameters number of nodes and number of connections after filtering to types contained in a given PGSL pattern. The decrease percentages are averaged for the searches in the 74 designs and separated for each of the five patterns we use in the experimental chapter (see Chapter 6).

Average decrease of ...	*Counter 1*	*Counter 2*	*State Machine*	*Encode/Decode Element 1*	*Encode/Decode Element 2*
Number of nodes	41.5%	70.4%	59.3%	37.5%	35.6%
Number of connections	66.6%	76.2%	62.9%	66.6%	59.9%

5.2 Creating the filtered design graph

The design graph of a given *Verilog* module constitutes the search space in which we aim to find matches with respect to a given PGSL pattern. A design graph consists of nodes with node types as labels and directed edges between these nodes. For a search with a specified PGSL pattern, not all of the nodes and connections in the design graph are relevant. For example, we do not care about nodes labeled as $xor if a pattern does not contain any nodes that can match an $xor. As we collect the node types that we want to match with the pattern in the parsing stage as "needed types" we can prune the search space. We accomplish this by iterating through all nodes contained in the original design graph and deleting all nodes that have a type that is not in the "needed types" set of the given pattern. Additionally, we delete all connections from and to the deleted nodes from the design graph's adjacency matrix. We call the resulting graph "filtered design graph" and use this graph in all following steps of the search as search space.

Filtering to the "needed types" does not guarantee that the search space gets smaller for every given design graph and PGSL pattern. In the worst case, when the design graph contains only nodes of types that are in the "needed types" set, the filtering cannot remove any nodes and connections. In that case the search space size remains the same after filtering. Nevertheless, our experiments (see Chapter 6) show that our initial filtering on average significantly reduces the number of nodes and connections. Table 5.1 shows the average decrease of the search space when filtering the design graph for 74 designs we processed in our experiments. For all patterns we reduce the number of nodes in a design on average by at least 35% and the number of connections on average by at least 59%.

Additionally, we use the filtering stage to create initial candidates for all single nodes in the pattern. While looping through the nodes of the design graph we store the IDs of nodes in the design graph grouped by each node type that appears in the pattern. Therefore, if we need to know which nodes in the design graph match a certain type, we do not have to loop through all nodes again, but rather solve this via lookup in the created map structure. We call this structure "candidates by type".

5.3 Hierarchical creation and combination of candidates

At the heart of the search operation we are solving a subgraph isomorphism problem. We aim to find the graphs specified by a pattern graph as subgraphs in a target design graph. The subgraph isomorphism problem for matching PGSL patterns in a design graph is special as PGSL patterns allow quantification of subgraphs. One pattern therefore can create multiple graphs for which we have to check if they match subgraphs of the target design graph. To visualize this, we can analyze a simple pattern containing a quantification as shown in Equation (5.2).

$add -> $mux >* -> $dff ; (5.2)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure 5.3: First three graphs the pattern $add -> $mux >* -> $dff defines. The quantification >* specifies that the subpattern $mux should be matched zero or more times in serial.

(0)
(3)
$add -> $mux >+ -> $dff
(1) (2) (4)

Figure 5.4: Example pattern, which we use to illustrate the traversal involved in the search process for this pattern. The numbers are the pattern-node IDs.

The pattern in Equation (5.2) specifies that the subpattern consisting of the node of type $mux should be matched zero or more times in serial indicated by the inner-operator '>' and the quantifier '*'. If we enumerate the graphs specified by this pattern, we find that this pattern stands for an infinite number of graphs: one without a $mux node and infinite graphs with chains of the node type $mux from chain-length one to infinity. We illustrate the different graphs that are defined by this pattern in Figure 5.3. With PGSL, a quantified subpattern can be used in other subpatterns which themselves can again be quantified. In total, enumerating all the different graphs and searching for them one after another is not a viable option.

Next, we consider structural characteristics of pattern graphs. Our pattern graphs contain composite nodes and single nodes. These composite nodes themselves contain composite nodes and single nodes and so on. Because of this nested and hierarchical structure with quantified subgraphs, we can not simply start from a node in the pattern graph, search for all the possible matches in the design graph and use simultaneous traversals in both pattern graph and design graph to find full subgraph matches. The search process has to be rather a procedure of solving smaller searches and combining them until matches for the whole pattern are found. Therefore, we use a divide-and-conquer approach starting at the top composite node. For each composite node, we first find candidates for the contained single nodes. These are the easiest to create and therefore, if we find no candidates for the single nodes, we can quickly terminate the search process. Next, we create candidates for the composite nodes that are contained in the composite node which themselves can again contain composite nodes and single nodes. Once all contained nodes in a composite node have candidates, we create candidates for this whole composite node by combing them with a custom CSP solver. This leads to a create-combine-move-up traversal in the pattern graph until no match for a subpattern can be found or candidates for the whole pattern (the top composite node) are created. We store all candidates in a table with unique IDs and use these IDs are as references in combined candidates. As a preview to the following subsection, we roughly illustrate the overall search procedure for an example pattern shown in Figure 5.4:

(a) Step into the top composite node (0) containing the whole pattern.

(b) Create candidates for all contained single nodes in (0). In this case we have the single nodes (1) and (4). Single-node (1) matches to cells of type $add in the design graph and single node (4) matches to cells of type $dff in the design graph. In addition to the type constraint, the candidate cells in the design graph which are candidates for single node (1) need to have an outgoing connection to a $mux cell. Similarly, the candidate cells in the design graph which are candidates for single node (4) need to have an incoming connection from a $mux cell.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

(c) Create candidates for all contained composite nodes in (0). Therefore, we step into the first (and only) composite node (3) in our pattern.

(d) Create the candidates for the subpattern contained in the this composite node (3). The subpattern is the single node (2), so we create candidates for it. Due to the serial-quantification and the adjacent single nodes (1) and (4) cells in the design graph that match the single node (2) need an incoming connection from either an \CO cell or $mux cell and an outgoing connection to either a $mux cell or a $dff cell.

(e) As the composite node (3) is serially-quantified, the candidates that we create in step (d) are candidates of a single stage of a serially-connected chain. We call them *stage candidates*. We combine them to serially-connected chains with our CSP solver. This creates candidates for the composite node (3) with quantification.

(f) All contained nodes in the composite node (0) now have candidates, therefore we combine them with a call to our CSP solver, with the connections (1)->(2) and (2)->(4) as constraints. The resulting candidates are candidates for the top composite node (0) and therefore results of our pattern search.

In the following subsections we discuss the methods and structures we use in our search-and-combine pattern-matching approach. A high-level view of this approach can be seen in Figure 5.5. We first present the creation of candidates and leave out the details concerning our CSP solver. This way, we can subsequently present the CSP solver as a whole with all features that we add in order to solve our specific search problem.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[5] C. Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

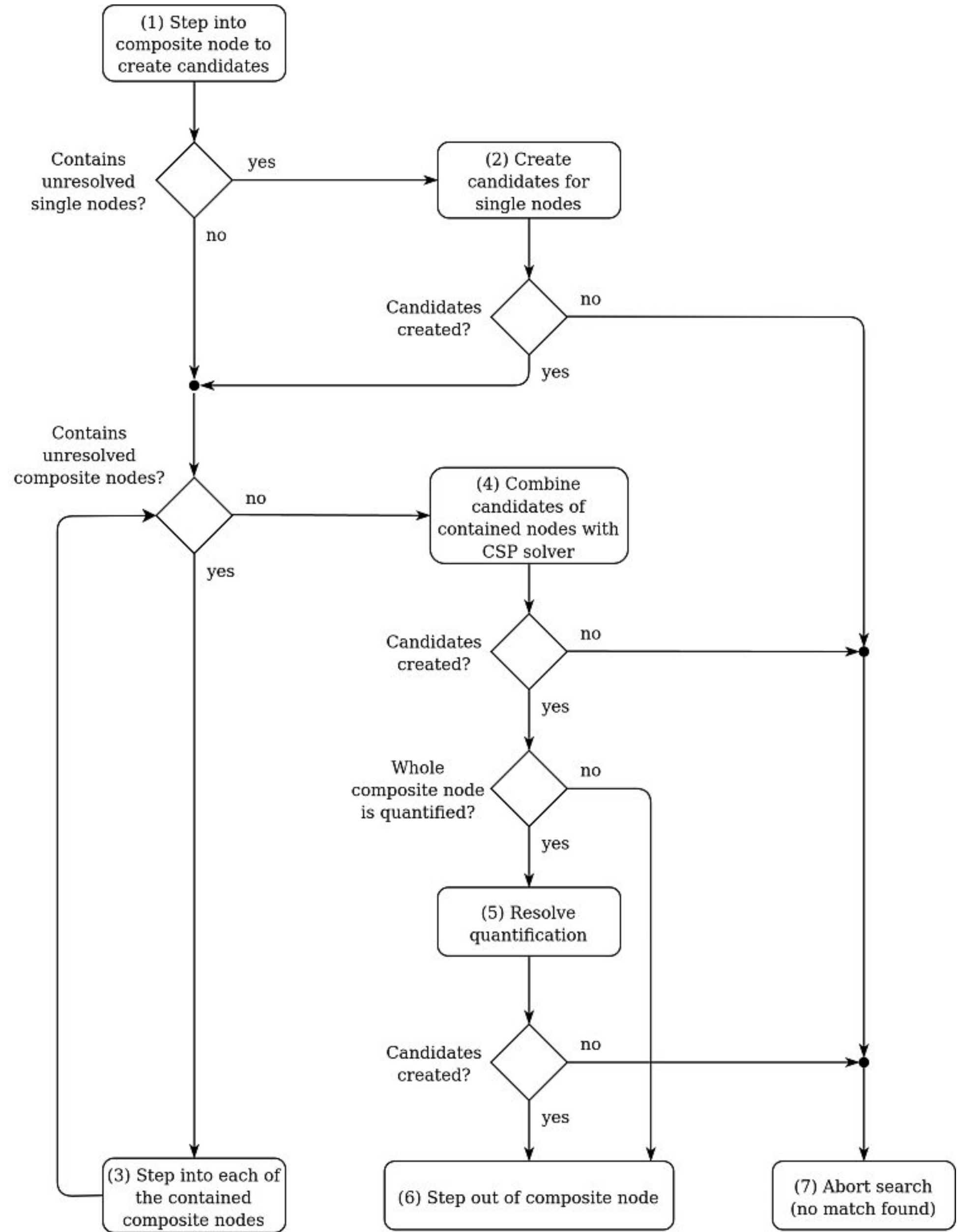


Figure 5.5: High-level view of our search-and-combine methodology. Starting with the top composite node we resolve a composite node by stepping into it (1). Single nodes are always resolved first (2). If the composite node contains other composite nodes, they are resolved next by recursively stepping into them (3). If all nodes of a composite node have been resolved we use our CSP solver to combine the candidates of the contained nodes to a candidate for the composite node (4). If the composite node is quantified, we next resolve the quantification (5). When a composite node is fully resolved and has an non empty set of candidates, we can step out of the composite node (6). In the case of the top composite node, stepping out at (6) terminates the search operation with the created candidates being the results of the search. If at any point in the search process an empty set of candidates for a node is created, the search terminates with no results (7). A similar flow chart can also be found in the PhD thesis of Krieg[5] as that work also discusses the algorithm that was developed in this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

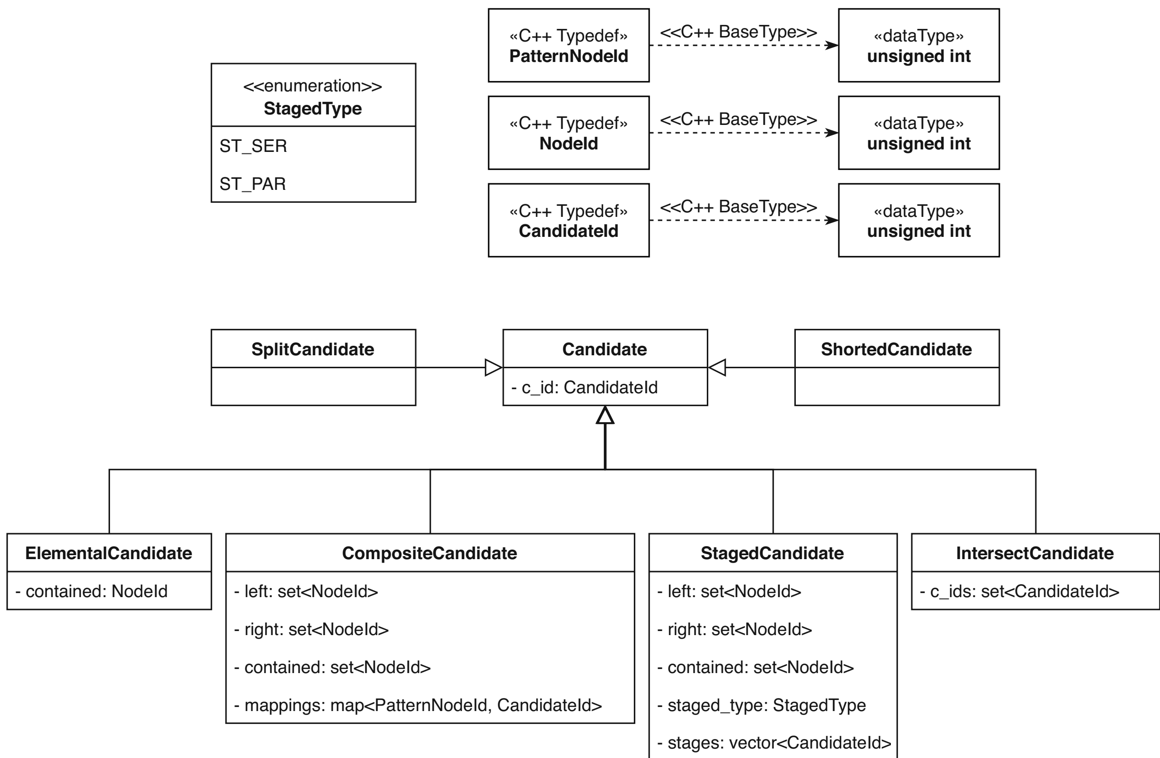


Figure 5.6: Types of candidates we use in our search operations. Only the important attributes of each class are shown. *Candidate* is the abstract superclass for all candidate types. *NodeId* is an ID of a node in the design graph, *PatternNodeId* is an ID of a node in the pattern graph and *CandidateId* (as variable *c_id*) is the unique ID we assign each candidate we create.

Figure 5.6 gives an overview of the classes in which we store candidates for pattern nodes. Each candidate type has its own class and is a subclass of the abstract class *Candidate*. We assign each candidate a unique candidate-ID, a candidates table stores the mapping between ID and candidate object. To save memory, a combined candidate only stores the candidate ID of its contained candidates. This way we can easily reuse candidates that are contained in multiple higher-level candidates. We describe the different candidate classes and their fields in the following subsections.

Creating candidates for single nodes

To create candidates for single nodes we first look at their node-type specification. A single node can match, depending on its specification in the PGSL pattern, one or a set of node types in the design graph. For example, $mux can match only to $mux nodes, while ($mux | $pmux) can match both $mux and $pmux nodes. The candidates-by-type structure we create during filtering to the filtered design graph gives us a first baseline for creating domains (list of candidates) for single nodes. To further reduce the domain for a single node we inspect the neighbors of this node in the pattern graph to create a neighborhood profile of the node in the pattern graph. We use a neighborhood profiler to loop through and check each of the type matching candidates from the candidates-by-type list and discard those that do not fit the profile. As the candidates for a single node can only contain exactly one node in the design graph, we store them in an object of class *ElementalCandidate*. By storing these single node matches in objects of a class that is a subclass of the abstract *Candidate* class, we can easily reuse them when combining to higher-level candidates.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Creating candidates for composite nodes

In order to create candidates for composite nodes, we first create candidates for all the contained pattern nodes. Once each contained node has candidates, our CSP solver is in charge of combining them to candidates for the whole composite node. Each combined candidate has to fulfill the constraints given by the adjacency of the nodes in the composite node. The resulting candidates are objects of the class *CompositeCandidate*. Each composite candidate contains a mapping structure (*mappings*) in which each pattern node contained in the composite node is assigned a candidate that matches the pattern specified by the pattern node. Additionally, compared to candidates for single nodes, a composite node contains more than one node from the design graph. Therefore, it is important for future combinations to know which of these design graph nodes are on the left and on the right side of the combined composite candidate and which design graph nodes are contained in the composite candidate. The fields *left*, *right*, and *contained* store the corresponding IDs of nodes in the design graph. If the composite node is not quantified (as always in the case of the top composite node of the pattern graph) no further action has to be taken. Otherwise, if the composite node is quantified, we need to resolve its quantification (see next two subsections).

Resolving serial quantification

To resolve serial quantification, we have to find chains of the composite candidates for the composite node in the design graph. In this context we call the composite candidates "stage candidates". For finding chains of the stage candidates connected in serial in the design graph, we incrementaly build up chains using these stage candidates. For simple composite nodes like ($mux) >+ this problem could be solved by starting at each of the stage candidates and performing DFS searches in the design graph. This method breaks down if the stage candidates contain more than one design graph node on their left or right side. Therefore, to have one uniform approach, we build up our chains with incremental calls to our CSP solver. The first call combines stage candidates with stage candidates to build chains of size two. Of course, the solver only combines two stages, if their graphs are serially connected in the design graph. The next call tries to build three-stage candidates with the two-stage candidates and the stage candidates. Each subsequent call aims to extend the existing chains by one. All chains of a length that fulfills the length interval specified with the quantification are added as candidate for the whole serially-quantified composite node. This procedure aborts once the upper limit of the quantification is reached (e.g. > {,3} specifies chains of maximal length three) or no further chain extensions are found. We store chains of stage candidates as objects of class *StagedCandidate* with the *staged_type* set to *ST_SER* ("staged serial"). The stages are stores in a list of the *C++ type vector*.

A special case is the search for serial-quantifications that allow a chain length of zero (e.g., the quantification ‘>*’ specifies chains of length zero or more). We interpret chains of length zero as "shorted" and add an own candidate type *ShortedCandidate* to the candidates of the composite node. Our CSP solver is able to handle this *ShortedCandidate* and act accordingly when combining in the composite node one layer above in the hierarchy.

Resolving parallel quantification

For resolving parallel quantification, we need to identify sets of stage candidates, that are parallel in the design graph. Parallelism cannot be determined by only inspecting the stage candidates, as parallelism is determined by a common

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

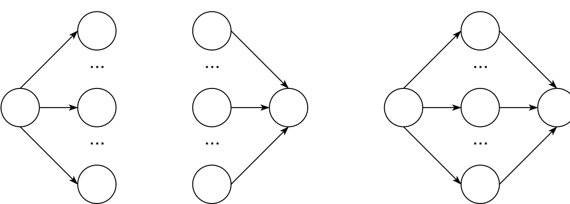


Figure 5.7: Visualizing the three cases for valid parallel quantification. The parallel-quantified nodes either need a common branch, a common sink, or both.

sink and/or branch as we illustrate in Figure 5.7. As an example, we analyze the pattern *Encode/Decode Element 2* which we created in Section 4.2 and we use in our experiments (see Chapter 6). In this pattern (see Equation (5.3)) the node $reduce_or constitutes the common sink for the parallel quantified composite node (\CO -> $eq) |+ .

$$\texttt{(\textbackslash CO -> \$eq) |+ -> \$reduce_or ;} \tag{5.3}$$

To combine the stage candidates in parallel, we take the candidates of the branch and sink nodes that are serially connected to the parallel-quantified composite node into account. We check which of the candidates of branch and/or sink nodes have connections to which stage candidates. We obtain a set of stage candidates for each branch/sink candidate, that constitutes the stage candidates that are in parallel in respect to this branch/sink candidate. These sets are stored in objects of class *IntersectCandidate*. The reasoning behind the name of this class will become clear when we discuss the CSP solver in depth in the next subsection. We use the pairs of branch/sink candidate and objects of class *IntersectCandidate* as constraining assignment in the CSP solver (we present details in the next subsection).

Candidates for branch and sink nodes must exist to enable the search algorithm resolve a parallel quantified node. Thus, at each hierarchy of a pattern parallel quantifications are always resolved last.

Again, quantifications that allow quantification of length zero are a special case (e.g., the quantification '|*' specifies the subpattern to be matched zero or more times in parallel). We interpret parallelism of size zero as split in the graph and add an object of the class *SplitCandidate* to the candidates of the composite node. Our CSP solver is able to handle this *SplitCandidate* and act accordingly when combining in the composite node one layer above in the hierarchy.

Candidate combination

An essential part of the search operation is a CSP solver that combines subpattern matches in the pattern hierarchy to bigger matches until a match for the top-level pattern has been created. We use the definition (which we introduced in Definition 3.1) for the simplest kind of CSP fitting our problem:

1. a finite set of variables $X = \{x_1, ..., x_n\}$,
2. a finite and discrete domain D_i of possible values for every variable $x_i \in X$ and
3. a finite set $C = \{C_1, ..., C_m\}$ of constraints on the variables of X.

CSPs have a state space. A *state* of a CSP is an *assignment* of values for some or all variables, $\{X_i = v_i, X_j = v_j, ...\}$. An assignment is *consistent* if it violates no constraint. A *complete assignment* is an assignment in which every variable is assigned a value from its domain. Finally, a solution of a CSP is a assignment that is consistent and complete.[6]

[6] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* 3rd. USA: Prentice Hall Press, 2009, pp. 202–239. ISBN: 0136042597.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

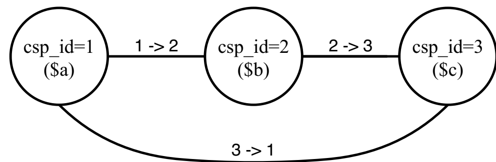


Figure 5.8: Constraint graph for combining candidates of the contained nodes in the pattern [[$a -> $b -> $c]]. Each of the vertices represents one of the contained single nodes of this pattern. Vertices connected by an edge have a common constraint which is labeled on the edge.

Two characteristics of our CSP problem lead us to the decision to implement our own CSP solver:

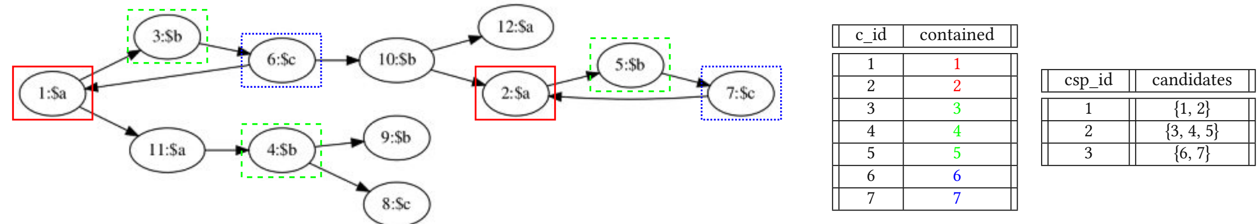
1. Our patterns allow quantifications with zero occurrence. For zero-occurence serial quantification we create a candidate that represents a through-connection in the subpattern ("shorted candidate"), for zero-occurrence parallel-quantification we create a candidate that splits the subpattern graph ("split candidate"). Picking such a candidate while solving the CSP dynamically changes the constraints on all other assigned candidates. For example, when we pick up a shorted candidate for a node in our subpattern, all connections to the node now become connections to the right neighbors of the node. This is because the adjacency constraints have to be pulled through the shorted candidate. Adding such a dynamic change of constraints on top of an existing CSP solver would not be efficient, as it is an integral part of our CSP problem.
2. Solving parallel quantification needs an extension to the classic CSP solving process. Again, we decide against externally attaching this to an existing CSP solver. To our knowledge existing CSP solvers do only support static domains (candidates that can be assigned to a node in the CSP). For parallel quantification, we need conditional assignments with intersection behavior: assigning a branch/sink node one of its candidates assigns the parallel quantified node candidates by intersecting the current list of candidates of the parallel quantified node with the candidates that are parallel in respect to the branch/sink node.

CSPs can be visualized as constraint graphs, where each vertex represents a variable and an edge between two vertices represents the constraint between those variables. For our CSP this is a suitable view as we can interpret the vertices as substitutes for the pattern nodes at the hierarchy at which we want to combine candidates. To get a grasp of the concept, we translate the candidate-combination problem of a simple example pattern (see Equation (5.4)) to a constraint graph.

[[$a -> $b -> $c]] ; (5.4)

This pattern does not contain any parallel quantified nodes, we present the extension for our CSP solver after this simple example. The pattern constitutes a top-level composite node which contains three single nodes. Our CSP solver is executed after each node contained in the pattern has a list of candidates. Figure 5.8 shows the constraint graph for the example pattern. We assign each of the vertices in the constraint graph a unique CSP-vertex ID (*csp_id*). The constraint graph visualizes the problem to solve: one CSP vertex for each contained single node, edges that show which vertices have a common constraint, and the adjacency constraints as labels of the edges. Each vertex represents a variable of the CSP, and their individual domains (possible values) are the set of candidates for each of the single nodes. We visualize the CSP setup for a fictional filtered design graph in Figure 5.9.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub



c_id	contained
1	1
2	2
3	3
4	4
5	5
6	6
7	7

csp_id	candidates
1	{1, 2}
2	{3, 4, 5}
3	{6, 7}

Figure 5.9: Visualizing the setup for our simple CSP. From left to right: filtered design graph, candidates table, CSP-vertices table with candidates per vertex. The node IDs (n_id) in the design graph are saved in candidates with unique candidate IDs (c_ids). As a result, each CSP vertex with an unique CSP ID (csp_id) has a set of candidates (its domain). In order to make the solving process easier to understand, this fictional created design graph has been created in a way so a candidate with *c_id=x* has a contained node candidate with *n_id=x*. This is not usual for real design graphs.

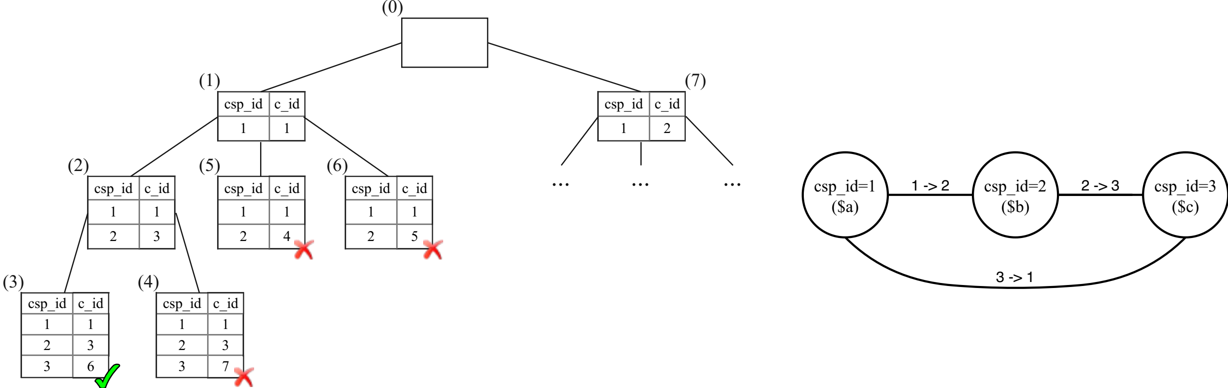


Figure 5.10: Tree that visualizes part of the backtracking search for our example CSP and the constraints as constraint graph. Starting from an empty assignment (0), first the vertex with *csp_id*=1 is assigned its first candidate (1). Subsequently the assignment is extended (2), (3). The traversal in this search tree is depth first. Backtracking is performed if a complete and consistent assignment (check-mark marks a solution; e.g., (3)) has been found, a constraint violation occurred (X marks a constraint violation; e.g., (4), (5), (6)) or a vertex has no more possible candidates to assign. After each backtrack the next possible candidate for the to be assigned vertex is added to the assignment.

The filtered design graph only contains nodes of types that appear in the example pattern. During creation of candidates for the single nodes contained in the example pattern, we assign each candidate an unique candidate ID (*c_id*) and store the resulting candidates in a candidates table. For our CSP, we create three vertices which each represent one single node and contain the set of candidates for the single node they represent. Additionally, we translate the adjacency constraints given by the adjacency between the single nodes contained in the composite node to an adjacency matrix of the CSP-vertices. To summarize, the call to our CSP solver has three parameters: (1) the filtered design graph, (2) the three CSP-vertices with sets of candidates, and (3) an adjacency matrix for the adjacency constraints.

In order to solve the CSP, we need a solving strategy. We use backtracking search, which is a depth-first search strategy. In this strategy, values (candidates) for one variable after another (represented by a CSP-vertex) are chosen to extend a CSP assignment, and backtrack occurs when

(1) a variable has no more value to assign, or
(2) a constraint between assigned variables is violated, or
(3) a complete and consistent assignment has been found.

We visualize the search strategy as a search tree in Figure 5.10. The process starts with an empty assignment and an empty set of solutions (step (0) in Figure 5.10). First, we pick the first unassigned vertex with *csp_id=1* and its first

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

candidate with *c_id=1*. We add the resulting pair to the assignment, forming an assignment of size one (step (1) in Figure 5.10). After each assignment extension the adjacency constraints have to be checked for the assignment. As our vertex with *csp_id*=1 has no adjacency with itself, this check succeeds as no constraints have to be checked. Next, we pick the next unassigned vertex with *csp_id*=2 and it's first candidate with *c_id*=3 and add the pair to the assignment (step (2) in Figure 5.10). This time, a constraint has to be checked as the vertex with *csp_id*=1 and the vertex with *csp_id*=2 have an adjacency (1->2, as seen in the constraint graph in Figure 5.8). Therefore, we inspect the candidates in the assignment and check if the subgraphs of the design graph they represent fulfill the adjacency constraint in the design graph. The candidate with *c_id*=1 contains the node with ID 1 in the design graph and the candidate with *c_id*=3 contains the node with ID 3 in the design graph. This check succeeds as the nodes 1 and 3 are connected as 1->3 in the design graph (see filtered design graph in Figure 5.9). Finally, we pick the last unassigned vertex with *csp_id*=3. We assign it its first candidate with *c_id*=6 and extend the assignment (step (3) in Figure 5.10). Subsequently, we check the constraints concerning this vertex (it needs an incoming connection from the candidate assigned to the vertex with *csp_id*=2 and an outgoing connection to the vertex with *csp_id=1*) and find that the candidate fulfills all constraints. As the assignment is complete and violates no constraint, we can add it as solution of our CSP problem. At a full assignment we backtrack, unassign the current candidate assigned to the last picked vertex with *csp_id*=3 and assign it its next candidate with *c_id*=7 (step (4) in Figure 5.10). The following constraint checks fail and we backtrack. As the vertex with *csp_id=2* has no next candidate we have to backtrack again. This assignment extension and backtracking terminates when the whole search space has been traversed. In our case, we obtain two solutions for the composite node we combined candidates for. We convert these solutions to objects of the class *CompositeCandidate* and as they are candidates for the top-level composite node the search terminates with two results for the pattern-matching problem. Although in this simple example only candidates of single nodes are combined by our CSP solver, the same principles apply when we combine for composite nodes, that do not only contain single node candidates.

Algorithm 5.1: First version of the backtracking search for combining candidates

Global variables in class CSPSolver: *unassigned_vertices*, *assignment*, *solutions*

```
1  Function CSPSolver::backtracking_search()
2      if assignment is complete then
3          add assignment to solutions
4          return
5      vertex ← next(unassigned_vertices)
6      csp_id ← csp_id(vertex)
7      foreach candidate_id in candidates(vertex) do
8          candidate ← candidates_table[candidate_id]
           /* add to assignment */
9          assignment[csp_id] ← candidate
           /* check constraints */
10         if not adjacency_okay(assignment, csp_id) then
11             erase assignment[csp_id]
12             continue
           /* new recursive call */
13         backtracking_search()
           /* restore assignment to state prior to assignment */
14         erase assignment[csp_id]
       /* add vertex back into unassigned_vertices */
15     insert vertex into unassigned_vertices
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

We implement our backtracking search in a class *CSPSolver* as recursive function. The first version of this backtracking search contains the concepts we discussed so far and can be seen as pseudo code in Algorithm 5.1. The call to the function *adjacency_okay* (line 10 in Algorithm 5.1) needs further explanation. This function collects and checks adjacency constraints after a vertex has been assigned one of its candidates. Depending on the type of candidate we assigned, we have to take different actions:

- ***ElementalCandidate, CompositeCandidate,* or *StagedCandidate***: We analyze the adjacency matrix of the constraint graph and examine each edge from or to the vertex which we assigned a candidate. An edge always connects two vertices. If both vertices of an edge have a candidate assigned in the current assignment, we add the edge and the two candidates to the list of adjacency constraints we have to check. In order for an adjacency constraint to be fulfilled, the two candidates have to be connected in the same way in the design graph. Each candidate represents a subgraph of the design graph and has a set of nodes of the design graph on its left and right side. Therefore we translate the adjacency constraint of the two vertices to one or more connections that have to exist in the design graph. We collect all these connections and confirm that they exist in the design graph. If any connection does not exist, the adjacency constraints check fails and the function returns *false*. Otherwise, the function returns *true*.

- ***SplitCandidate***: A split candidate splits the constraint graph at the position of the vertex it was assigned. No constraints have to be checked, the function returns *true*.

- ***ShortedCandidate***: A shorted candidate implies that the vertex in the constraint graph is now a through-connection. Therefore we pull all connections from and to the vertex through to gain a list of edges we have to check. We transform this list of edges to connections that have to exist in the design graph. Again, if any connection does not exist in the design graph, the adjacency constraints check as a whole fails, and the function returns *false*. Otherwise, the function returns *true*.

This first version of our backtracking search (see Algorithm 5.1) does not cover the handling of candidates for parallel-quantified pattern nodes. As we discuss in Section 5.3, candidates for parallel-quantified nodes are not independent, as their adjacent sink and branch nodes constrain the set of stage candidates that can be considered in parallel. For a simple pattern as seen in Equation (5.5), the node `$a` is the branch node and `$c` the sink node for the parallel-quantified node `$b |+`

$$\texttt{\$a -> (\$b) |+ -> \$c} \tag{5.5}$$

Let us assume that the parallel-quantified node has the stage-candidate set {1,2,3,4,5}, the branch node a candidate with *c_id*= 6 and the sink node a candidate with *c_id*= 7. Furthermore, during resolving the quantified-parallel node we determine that the following connections exist 6 -> {1,2,3} and 7 -> {2,3,4}. The only consistent assignment containing the candidate with *c_id*= 6 as branch and the candidate with *c_id*=7 as sink is the intersection of the two stage candidate sets: {2,3}. To represent this concept, we introduce different types of CSP vertices (an overview can be seen in Figure 5.11) and extend our CSP solver and its backtracking search.

An object of class *FreeVertex* represents a pattern node for which the assignment can be extended without influencing other CSP vertices. It only contains the domain of the vertex (the candidates of the pattern node it represents). This is the kind of CSP vertex we use in the first version (see Algorithm 5.1) of our backtracking search. An object of the class *ConstrainingVertex* is a CSP vertex for a sink or branch node. It contains a structure *candidates_implications* that stores

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

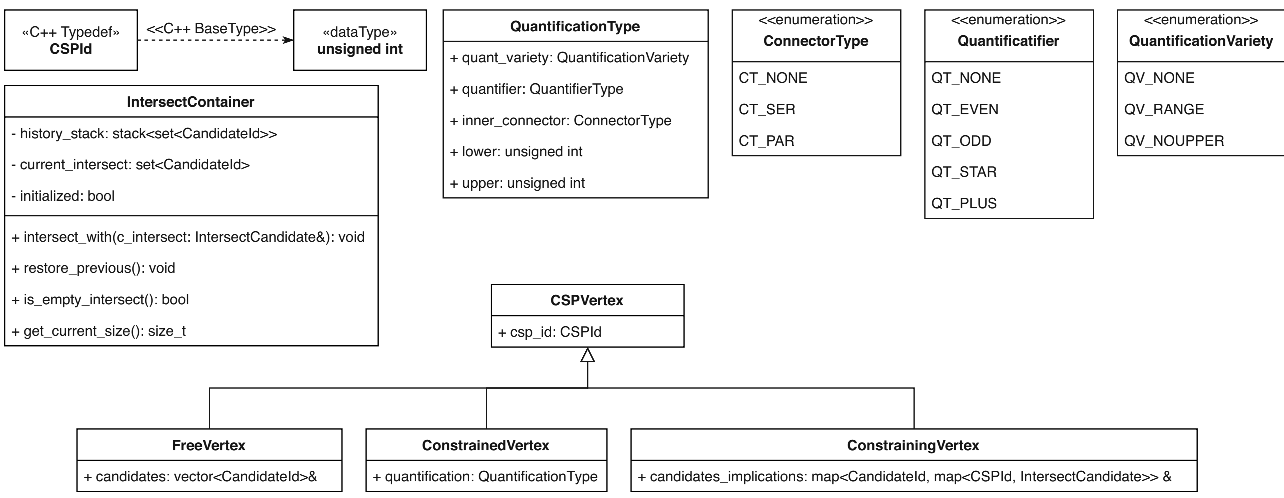


Figure 5.11: Classes and types involved in setting up a CSP backtracking search. Only the important attributes and methods of each class are shown. *CSPVertex* is the abstract superclass for all other CSP-vertices. The *IntersectContainer* class plays a core role in finding the common parallel candidates (intersection) for parallel-quantified nodes, if the node is constrained by more than one branch and/or sink.

the constraining assignment that influences the candidates of a constrained vertex. An object of class *ConstrainedVertex* represents the parallel-quantified node and stores the parallel-quantification information. Assigning a branch or sink node a candidate influences the state of the constrained vertex. The state is the set of candidates that are considered to be parallel to form a consistent assignment at a given point in the CSP solving process. In order to track the state of a constrained vertex, we do not include the candidates for the parallel-quantified candidates in the assignment in our backtracking search. A variable that represents a parallel-quantified node in our CSP solver is represented by an object of class *IntersectContainer*. This class offers additional properties we need to keep track of the state of a constrained node: applying a set of candidates and intersecting and restoring the state to a previous state.

Every time we assign a constraining vertex a candidate during backtracking search, its affiliated set of stage candidates for a parallel-quantified node is applied as an intersection to the intersect container that represents the parallel-quantified node. For our example pattern in Equation (5.5), when assigning the branch constraining-vertex the candidate with *c_id*=6 the intersect container is initialized with the candidate set {1,2,3}. When we assign the sink constraining vertex with *c_id*=7 the candidate set {2,3,4} is applied as intersection. The resulting state of the intersection container with state {1,2,3} intersected with the set {2,3,4} results in a new state of {2,3}. For backtracking, each intersect container stores its state history in a stack. Therefore, when unassigning a constraining vertex we can restore the state of the intersect container to before the assignment. When an assignment in our backtracking search is full and consistent, the state of each intersect container is added to the assignment as object of class *IntersectCandidate*. A intersect candidate is transformed to an object of class *StagedCandidate* with *staged_type* set to *ST_PAR* ("staged parallel") in a post-processing step. This way we end up with solutions that contain assignments for each of the CSP vertices and can assemble a composite candidate.

Through experimentation we detect another constraint, that is not yet covered in out CSP solver: the all-different constraint. The all-different constraint represents the notion that a valid combined candidate cannot contain a node from the design graph more than once. Without this constraint, we would gain false matches in the design graph. For example, in a pattern `$a -> $b -> $a` the two single nodes that each match the type $a can have overlapping sets of candidates. With the given pattern we specified that we want to match two distinct single nodes of type $a. Therefore,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

assigning each of the vertices the same node from the design graph does not lead to a match, that was specified by the pattern. Therefore, we can conclude that a valid assignment in our CSP solver cannot contain the same node from the design graph more than once. Thus, after each assignment extension, we have to check if the sets of contained nodes from the design graph of each candidate in the assignment do not intersect. If the all-different constraint is violated, the assignment is not consistent, which leads to a backtrack.

We present the pseudo code for our final backtracking search in Algorithm 5.2. In comparison to our first version (see Algorithm 5.1), this final version contains the two discussed additional concepts: handling candidates for parallel quantifed nodes and the all-different constraint. Picking a free vertex or a constraining vertex trigger different actions. For free vertices we only have to check the adjacency constraints and all-different constraint, while picking a constraining candidate additionally implies intersecting in intersection containers and checking the resulting intersects.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 5.2: Final version of the backtracking search for combining candidates

Global variables in class CSPSolver: *unassigned_vertices*, *assignment*, *intersect_containers*, *solutions*

```
1  Function CSPSolver::backtracking_search()
2      if assignment is complete then
3          add assignment and intersect_containers to solutions
4          return
5      vertex ← next(unassigned_vertices)
6      csp_id ← csp_id(vertex)
7      if vertex is FreeVertex then
8          foreach candidate_id in candidates(vertex) do
9              candidate ← candidates_table[candidate_id]
               /* add to assignment */
10             assignment[csp_id] ← candidate
               /* check constraints */
11             if not alldiff_okay(assignment, csp_id) then
12                 erase assignment[csp_id]
13                 continue
14             if not adjacency_okay(assignment, csp_id) then
15                 erase assignment[csp_id]
16                 continue
               /* new recursive call */
17             backtracking_search()
               /* restore assignment to state prior to assignment */
18             erase assignment[csp_id]
19     else if vertex is ContrainingVertex then
           /* loop through the candidates_implications for this vertex */
20         foreach (candidate_id, intersect_map) in candidates_implications(vertex) do
21             candidate ← candidates_table[candidate_id]
               /* add to assignment */
22             assignment[csp_id] ← candidate
               /* check constraints */
23             if not alldiff_okay(assignment, csp_id) then
24                 erase assignment[csp_id]
25                 continue
26             if not adjacency_okay(assignment, csp_id) then
27                 erase assignment[csp_id]
28                 continue
               /* check intersects, applies intersects if non empty and above lower bound */
29             if not intersections_okay(intersect_map) then
30                 erase assignment[csp_id]
31                 continue
               /* new recursive call */
32             backtracking_search()
               /* restore assignment to state prior to assignment */
33             erase assignment[csp_id]
               /* restore intersect_containers to state prior to assignment */
34             foreach elemement in intersect_map do
35                 container_id ← element.first
36                 intersect_containers[container_id].restore_previous()
           /* add vertex back into unassigned_vertices */
37         insert vertex into unassigned_vertices
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

5.4 Result post-processing and visualization

The final candidates for a pattern need to be post-processed in order to be visualized and saved for *Yosys* plugins that want to build on the result of the pattern search. This post-processing consists of three parts:

(1) untangling the candidate structures to graphs,

(2) eliminating duplicate, and

(3) storing and visualizing the result graphs, and creating a search summary.

Candidates of the top-level composite node represent subgraphs of the design graph. Nevertheless, top-level candidates are a product of the match-and-combine process and therefore are nested structures with references (by ID) to other candidates which themselves can have references to other candidates, and so on. In sum, a top-level candidate is not a graph structure, that just contains the subgraph in the design graph as list of design graph node IDs and an accompanying adjacency matrix. Therefore we recursively collect the node IDs and adjacency information contained in the candidates belonging to a top-level composite candidate and build up a simple graph structure.

Our search algorithm can create duplicate result graphs for patterns that have a built in possibility for permutations in the matching process. An example is the pattern `($a || $a) -> $b`. It contains two nodes of type $a in parallel connected to a common sink node of type $b. Let's assume that there exist two nodes (node ID 1 and 2) in the design graph that match the constraints type=$a and a connection to a common node of type $b. Our search process will create two candidates:

(1) one with the candidate for *node ID=1* assigned to the first $a and the candidate for *node ID=2* to assigned the second $a, and

(2) one with candidate for *node ID=2* assigned to the first $a and the candidate for *node ID=1* to assigned the second $a.

These two candidates create the same subgraph of the design graph, therefore our solving process created a duplicate. In order to remove these permutation-based duplicates, we hash all graphs and check for hash collision. If a hash collision occurs, we compare the graphs and if necessary discard duplicates.

Finally, our result graphs are saved in tables that allow following *Yosys* plugins access. In addition, we save all subgraphs as graphs in the *dot language*,[7] as pictures and produce a HTML search summary that lists the results per module and shows the result graph when hovering over the list of contained nodes for a result graph.

[7] *The DOT Language*. https://www.graphviz.org/doc/info/lang.html.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 6

Experiments and results

To illustrate the efficiency and wide-ranging applicability of the subcircuit pattern search methodology introduced in Chapter 5, we conduct a large-scale experiment using *Verilog* HDL designs available from *OpenCores*.[1] We synthesize the collected designs, conduct pattern searches with five selected PGSL patterns, and post-process the search results. The five patterns cover three distinct functional primitives and demonstrate all major features of PGSL (loops, serial quantification, and parallel quantification). Overall, our experiment covers 74 designs of different sizes: the smallest design contains 28 cells and 58 connections after synthesis, the biggest design contains 104785 cells and 158657 connections after synthesis. We conduct the experiment on an Intel® Core™ i7-8750H CPU at 2.20 GHz and 16 GB of physical memory. Results indicate that our search methodology is able to process even large designs with reasonable runtimes (average search time 2.8 seconds, maximal search time 141.5 seconds). We summarize the patterns we use for our experiments in Table 6.2. Table 6.4 shows the experimental results.

6.1 Experimental setup

With our experimental setup we synthesize and search in the designs from *OpenCores* and post-processes the results. As *OpenCores* HDL projects do not follow a common file structure, do not offer synthesis information in a common format and occasionally have vendor specific dependencies (e.g., *Xilinx* vendor libraries) errors in individual stages can occur. Therefore we only were able to use 74 of the existing *Verilog* designs from *OpenCores*. We collect results in a database.

Synthesis

To synthesize we use a custom *Yosys* script for each of the HDL designs. An example for the aes_core design (design at line 6 in Table 6.4) is shown in Listing 6.1. First, *Yosys* loads all *Verilog* files with its *Verilog* frontend using *read_verilog* (line 1-6). Next, *Yosys* is instructed to perform a conservative RTL synthesis using the command *prep* (line 7). With the parameter *-auto-top* the hierarchy of the design resolves automatically, therefore it does not matter in which order

[1] *OpenCores*. https://opencores.org/.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Listing 6.1: Example *Yosys* synthesis script for the aes_core design.

```
read_verilog aes_core/trunk/rtl/verilog/aes_cipher_top.v
read_verilog aes_core/trunk/rtl/verilog/aes_inv_cipher_top.v
read_verilog aes_core/trunk/rtl/verilog/aes_rcon.v
read_verilog aes_core/trunk/rtl/verilog/aes_sbox.v
read_verilog aes_core/trunk/rtl/verilog/aes_inv_sbox.v
read_verilog aes_core/trunk/rtl/verilog/aes_key_expand_128.v
prep -auto-top
flatten
write_verilog synthesis/synthDesigns/aes_core_synthesizedDesign.v
```

the individual *Verilog* files are read. Finally, the resulting design is flattened (line 8) and written to a file for further processing (line 9).

Search

To prepare a design for the pattern search, it is first transformed into a design graph according to the design graph abstraction, which we elaborate in Section 3.1. Next, successively we search for all five selected PGSL pattern in the design's design graph. We measure the runtime of each search operation as difference of the operating system's timestamps before and after the search operation. We use this measurement as a metric of the performance of a search with a given design and a PGSL search pattern. If the search returns matches for the given pattern, the results are saved as subgraphs of the design graph. We save these subgraphs both internally and as human-readable graphs for further processing. Additionally, we create statistics concerning the original design graph, the filtered design graph, and the search results during the search operation. We use these statics to understand the performance of the search operation for a given design and a PGSL pattern. The list of statistics is shown in Table 6.1.

For this experiment we use the PGSL patterns we presented in Chapter 4. These patterns allow us to search the designs for functional kernels of common functional primitives: counters, state machines, and elements of encoders and decoders. In this context, we use functional kernel as term for the core characteristics that make up the functionality of a functional primitive (e.g., a counter is a register, that is incremented and the incremented value is fed back to the register). Finding these functional primitives is of relevance for verification and reverse engineers: identified state machines can be optimized, finding memory decoders/encoders can be used for memory identifications, and identified counters can be used for security evaluations (e.g., hardware Trojan triggers are known to use counters). In addition, these patterns demonstrate the prominent features of PGSL: loops, serial quantification, and parallel quantification. Finally, the patterns we choose differ in complexity, which we use to demonstrate the efficiency of our search method.

Table 6.1: Statistics collected by the search plugin during a search operation.

Related to	Item	Description
Design graph and filtered design graph	num_cells	Total number of cells in the design graph.
	num_connections	Total number of connections in the design graph
	cell_count_map	Number of cells for each cell type in the design graph
	avg_degree	Average in+out-degree of the nodes in the design graph
	max_degree	Maximal in+out-degree of a node in the design graph
Search results	num_matches	Number of search results
	min_cells	Number of cells of the result with the least cells
	max_cells	Number of cells of the result with the most cells
	min_connections	Number of connections of the result with the least connections
	max_connections	Number of connections of the result with the most connections
	max_degree	Maximum in+out-degree for a node in all search results

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 6.2: Overview of the five patterns used to search in the *Verilog* designs obtained from *OpenCores*.

<table>
<tr><th>#</th><th>Function</th><th>PGSL pattern and schematic</th><th>Description</th></tr>
<tr><td></td><td>Global Macros</td><td>@MUX = $mux | $pmux ;
@REG = $dlatch | $dff | $adff ;</td><td>Global macros that can be used in other PGSL patterns. @MUX represents different types of multiplexers, @REG represents register memory elements.</td></tr>
<tr><td>1</td><td>Counter 1</td><td>\CO -> [[$add -> @MUX >*-> @REG]] ;</td><td>A counter that is incremented by a constant value (\CO). The path between the addition cell ($add) and the register element (@REG) can contain a multiplexer chain that implements the counter's control path.</td></tr>
<tr><td>2</td><td>Counter 2</td><td>[[$add -> @MUX >*-> @REG]] ;</td><td>A counter that is similar to Counter 1, but also matches to non constant increment counters, e.g., the increment is by a value stored in a register. By using two counter patterns a verification or design engineer can easily discover which counters that are incremented by constants versus counters that are incremented by non constant values.</td></tr>
<tr><td>3</td><td>State Machine</td><td>[[@REG -> (($mux | $logic_not) |* || $eq |+) -> $pmux -> @MUX >*]] ;</td><td>A state machine where the state register element (@REG) is compared to the state machines' possible state values. The comparisons are done by comparators ($eq), multiplexer cells ($mux) or inverters ($logic_not). Using an inverter ($logic_not) as comparator, is a common optimization in Yosys for comparing to logical 0. The comparisons are fed to a parallel multiplexer cell ($pmux) followed by an optional multiplexer chain to complete the feedback path that is involved in calculating the next state value.</td></tr>
<tr><td>4</td><td>Encode/Decode Element 1</td><td>(\CO -> $eq) |+ -> @MUX ;</td><td>An element of an encoder/decoder. In encoder or decoder comparisons to constants are performed and fed to multiplexers to control following operations. As decoders/encoders are used to access memory, we also find structures that represent memory with this pattern, for example lookup tables.</td></tr>
<tr><td>5</td><td>Encode/Decode Element 2</td><td>(\CO -> $eq) |+ -> $reduce_or -> @MUX ;</td><td>An element of an encoder/decoder, similar to Encode/Decode Element 1. This version matches to a variant created by a Yosys optimization. Yosys adds an extra $reduce_or cell for comparisions that check if a value is in a set of values.</td></tr>
</table>

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

As verification or reverse engineers might want to differentiate between counters that are incremented by constants and counters that are incremented by non-constant values, we use two counter patterns: one that only matches constant-increment counters, and another that in addition matches non-constant-increment counters. To cover a *Yosys* optimization for parallel comparisons, we search for encode/decode elements in two variants. The second variant matches parallel comparisons, that check if a value is in a set of values. *Yosys* optimizes these parallel comparisons with an additional "reduce or" operation, as only one of the comparisons has to be true. With only the first "normal" decode/encode pattern, we would not be able to find these special, optimized decode/encode elements. For state machine identification we use one unified pattern. We summarize the five used patterns, their schematic view and short description in Table 6.2. The cell types in the PGSL patterns are internal cell types of *Yosys* (e.g., $mux is an internal representation for a multiplexer).

Post-processing

With our search method we search for functional kernels of functional primitives in HDL designs. Verification and reverse engineers might be interested in the full functional primitive including the signals and inputs that influence the functional primitive. For example, it is of interest which inputs or signals are compared by an encode/decode element. To demonstrate that the search results can be post-processed for such an usage, we implement post-processing of search results. The post-processing extracts modules from the search results including the input cone of all the cells included in the search result. In this context, an input cone of a cell is the tree of wires and cells at the input of the cell in the original design. The input cone starts at cells and wires of the search result subcircuit and ends at primary inputs of the design graph or at outputs of registers. The post-processing extracts the subcircuit of the original design and its input cone into a *Verilog* module. Similar to the search operation, we also measure the runtime of the post-processing operation.

Additionally, post-processing can be used to eliminate unintended duplicate results by merging search results. For example, the feedback path of a counter can be formed by two different multiplexer chains, as the feedback path is not a chain but rather a tree. For such a case, the search operation returns two distinct results, post-processing creates a module for each of them, but discovers the duplicate and only outputs one module. This post processing step is described in detail by Krieg.[2]

6.2 Experimental results

Table 6.4 gives an overview of the experiments with 74 *Verilog* HDL designs, that we collected from *OpenCores* and the five selected PGSL search patterns. Each line represents a design from *OpenCores*. We order the designs alphabetically and show the category in which they are listed at *OpenCores*. We use abbreviations for the categories, the mapping from abbreviation to full name is shown in Table 6.3. The columns *cells* and *conn* show the number of cells and connections of the design graph of each synthesized and flattened design. The subsequent five columns show the results for each of the five patterns. *S* represents the number of results for the search operation, *P* represents the number of remaining results

[2] C. Krieg. “Pattern-Based Hardware Trojan Characterization for Design Security Assessment”. PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 6.3: Abbreviations and full names used for the *OpenCores* categories.

Abbreviation	Full name
arith	arithmetic_core
co_proc	coprocessor
comm	communication_controller
crypto	crypto_core
dsp	dsp_core
ecc	ecc_core

Abbreviation	Full name
mem	memory_core
proc	processor
soc	system_on_chip
sys_ctrl	system_controller
test	testing/verification
video	video_controller

after post-processing. For both of those operations we also present the runtime in seconds, denoted as *t(S)* and *t(P)*. For each search and post-process operation pair we define a timeout of 20 minutes. Timeouts are marked as "Timeout". The last column *t* of the table shows the total search and post-processing time for each design as the sum of all t(S) and t(P). In the next subsections we present the results with illustrative examples and exceptional cases. Results that we cover in the next subsections are marked bold in the overview Table 6.4.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

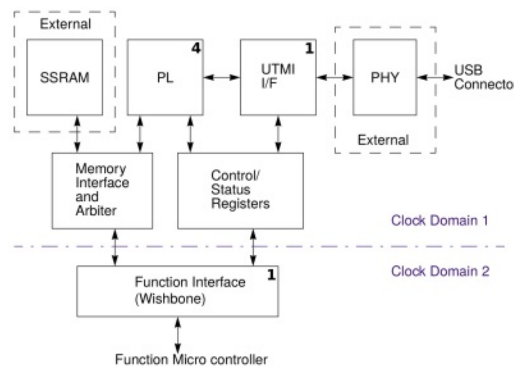
Table 6.4: Overview of the experimental results. For each pattern, S denotes the number of search results and P the number of modules after post-processing. $t(S)$ and $t(P)$ are the associated processing times. The last column t sums up the total processing time for each design. Results marked in bold are discussed in depth in the following sections.

#	Design	Category	Cells	Conn	**Counter 1**				**Counter 2**				**State Machine**				**Encode/Decode Element 1**				**Encode/Decode Element 2**				t[s]
					\CO -> [[$add -> @MUX >* -> @REG]] ;				[[$add -> @MUX >* -> @REG]] ;				[[@MEM -> (($mux | $logic_not) |* || $eq |+) -> $pmux -> @MUX >*]] ;				(\CO -> $eq) |+ -> @MUX ;				(\CO - > $eq) |+ -> $reduce_or -> @MUX ;				
					S	P	t(S)[s]	t(P)[s]	S	P	t(S)[s]	t(P)[s]	S	P	t(S)[s]	t(P)[s]	S	P	t(S)[s]	t(P)[s]	S	P	t(S)[s]	t(P)[s]	
1	8bit_vedic_multiplier	arith	533	926	0	0	0.023	0.019	0	0	0.022	0.016	0	0	0.021	0.015	0	0	0.021	0.017	0	0	0.022	0.015	0.19
2	a-z80	proc	327	778	0	0	0.004	0.006	0	0	0.004	0.006	0	0	0.005	0.005	0	0	0.004	0.005	0	0	0.004	0.005	0.05
3	ac97	sys_ctrl	1609	2867	32	23	2.475	5.386	32	23	2.366	5.416	0	0	0.076	0.029	33	33	2.616	7.623	0	0	0.02	0.027	26.03
4	aes-128_pipelined_encryption	crypto	104785	158657	0	0	1.873	0.592	0	0	1.919	0.588	0	0	3.061	0.575	Timeout				0	0	0.917	0.673	10.2
5	aes-encryption	crypto	104324	156451	0	0	0.818	0.66	0	0	0.945	0.582	0	0	1.311	0.565	Timeout				0	0	0.919	0.582	6.38
6	aes_core	crypto	10810	16294	2	2	0.515	2.243	2	2	0.521	2.226	0	0	0.295	0.151	21	21	141.457	66.29	0	0	0.21	0.185	214.09
7	aes_highthroughput_lowarea	crypto	5020	10289	2	2	0.463	2.328	2	2	0.437	2.334	1	1	0.454	1.315	35	35	2.958	38.404	0	0	0.119	0.196	49.01
8	aic1106_avalon_ip	comm	134	234	1	1	0.118	0.024	1	1	0.114	0.025	0	0	0.014	0.002	23	23	1.404	0.618	0	0	0.014	0.002	2.33
9	apbtoaes128	arith	1413	2561	3	3	0.483	0.805	3	3	0.507	0.837	2	2	0.555	0.615	35	35	2.814	12.145	22	12	2.243	3.355	24.36
10	bubblesortmodule	other	1416	2847	0	0	0.017	0.022	0	0	0.018	0.025	0	0	0.018	0.025	0	0	0.017	0.025	0	0	0.017	0.025	0.21
11	ca_prng	arith	565	856	0	0	0.017	0.007	0	0	0.016	0.007	**32**	**1**	5.431	0.625	32	32	3.533	2.042	0	0	0.016	0.007	11.7
12	des	crypto	1172	1682	0	0	0.033	0.017	0	0	0.027	0.024	0	0	0.023	0.016	0	0	0.024	0.016	128	8	12.674	5.027	17.88
13	dpll-isdn	arith	101	151	**5**	**4**	0.44	0.081	**5**	4	0.444	0.078	0	0	0.017	0.003	1	1	0.173	0.02	0	0	0.015	0.002	1.27
14	ezusb_io	comm	130	213	1	1	0.13	0.022	1	1	0.111	0.022	0	0	0.012	0.002	0	0	0.012	0.002	0	0	0.013	0.002	0.33
15	fast_antilog	arith	137	203	0	0	0.016	0.002	0	0	0.015	0.002	0	0	0.016	0.002	1	1	0.659	0.037	0	0	0.016	0.003	0.77
16	fast_log	arith	115	162	0	0	0.012	0.002	0	0	0.012	0.002	0	0	0.012	0.002	**2**	**2**	0.293	0.045	**13**	**1**	1.07	0.095	1.54
17	fpu	co_proc	1286	2253	0	0	0.108	0.03	0	0	0.107	0.028	0	0	0.057	0.023	34	34	2.849	11.252	14	8	1.085	1.955	17.49
18	freq_div	other	75	134	0	0	0.003	0.003	0	0	0.002	0.002	0	0	0.003	0.003	6	6	0.375	0.084	0	0	0.002	0.002	0.48
19	ftdi_wb_bridge	comm	336	629	17	2	**1.579**	0.287	9	2	**0.944**	0.19	2	2	0.566	0.158	11	11	1.027	0.525	1	1	0.141	0.061	5.48
20	gng	arith	766	1455	0	0	0.041	0.011	0	0	0.043	0.011	0	0	0.043	0.011	2	2	3.16	0.396	0	0	0.038	0.012	3.77
21	gost28147-89	crypto	661	1007	1	1	0.164	0.133	1	1	0.136	0.125	0	0	0.025	0.014	11	11	1.783	1.14	8	1	0.872	0.266	4.66
22	i2c	comm	434	817	0	0	0.017	0.006	0	0	0.022	0.013	2	2	0.719	0.163	15	15	1.432	0.996	8	5	1.307	0.303	4.98
23	i2cslave	comm	294	542	**6**	**2**	0.57	0.119	**6**	**2**	0.552	0.14	1	1	0.384	0.084	32	32	2.262	1.428	5	5	0.466	0.257	6.26
24	ima_adpcm_enc_dec	dsp	283	417	0	0	0.006	0.004	**3**	**2**	0.253	0.08	0	0	0.008	0.004	2	2	0.994	0.094	0	0	0.006	0.004	1.45
25	jt51	co_proc	8555	13437	15	13	**2.328**	17.547	41	37	**4.49**	46.727	1	1	0.83	1.428	102	102	**33.269**	123.709	307	13	35.201	97.632	363.16
26	linkruncca	dsp	264	522	4	4	0.364	0.216	4	4	0.345	0.219	0	0	0.025	0.007	5	5	0.364	0.259	3	2	0.247	0.149	2.19
27	m32632	proc	7318	13105	11	8	77.144	10.497	15	14	**78.19**	18.468	3	3	1.936	3.655	188	188	**20.835**	409.376	88	39	9.506	73.091	702.7
28	md5_pipelined	crypto	1810	3300	0	0	0.097	0.034	0	0	0.125	0.033	0	0	0.096	0.036	0	0	0.097	0.034	0	0	0.097	0.035	0.68
29	mem_ctrl	sys_ctrl	1972	6553	3	3	0.9	1.131	4	4	0.806	1.177	1	1	1.847	0.638	70	70	5.437	53.278	110	69	47.199	62.245	174.66
30	minsoc	soc	2821	5445	9	6	4.356	6.034	11	7	4.44	7.796	3	3	2.48	3.427	94	94	9.449	254.448	49	32	6.713	40.647	339.79
31	mips32r1	proc	1910	3452	1	1	0.495	0.435	3	2	0.633	0.937	0	0	0.246	0.03	32	32	3.395	11.862	48	18	5.053	6.924	30.01
32	mips789	proc	805	1440	3	2	0.278	0.276	4	2	0.345	0.3	1	1	0.128	0.149	57	57	4.658	12.097	50	41	5.404	8.045	31.68
33	mips_16	proc	194	329	1	1	0.101	0.06	2	1	0.154	0.049	0	0	0.006	0.003	**12**	**12**	0.813	0.396	**2**	**2**	0.352	0.094	2.03
34	mmcfpgaconfig	other	187	287	2	2	0.222	0.087	2	2	0.228	0.085	1	1	0.278	0.073	8	8	0.664	0.405	1	1	0.119	0.057	2.22
35	navre	proc	790	1531	7	5	0.929	1.007	11	5	1.187	1.064	1	1	0.299	0.173	71	71	5.677	22.617	51	33	6.214	6.785	45.95
36	nextz80	proc	1324	2537	1	1	0.734	0.189	1	1	0.807	0.172	**5**	**1**	1.145	0.39	153	153	10.013	47.345	250	141	20.112	42.787	123.69
37	pci	sys_ctrl	2842	5740	19	19	2.335	10.94	19	19	2.305	10.637	4	4	0.808	2.817	75	75	6.493	47.35	31	23	2.85	12.432	98.97
38	pid_controller	dsp	1666	3530	0	0	0.027	0.046	0	0	0.027	0.047	1	1	0.244	0.424	30	30	2.23	15.391	6	6	0.565	2.385	21.39
39	present_encryptor	crypto	597	880	1	1	0.168	0.081	1	1	0.146	0.091	1	1	0.353	0.078	17	17	3.117	1.406	0	0	0.02	0.011	5.47
40	qspiflash	comm	1671	3428	0	0	6.835	0.02	0	0	6.924	0.021	0	0	6.656	0.022	661	661	42.266	928.248	14	14	1.553	3.09	995.63
41	random_pulse_generator	other	29	48	0	0	0.001	0.001	0	0	0.001	0.001	0	0	0.001	0.001	0	0	0.001	0.001	0	0	0.001	0.001	0.01

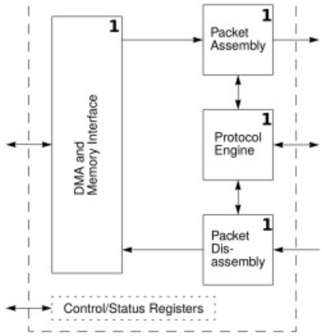
TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Table 6.4: - continued from previous page

#	Design	Category	Cells	Conn	Counter 1 \CO -> [[$add -> @MUX >* -> @REG]] ;				Counter 2 [[$add -> @MUX >* -> @REG]] ;				State Machine [[@MEM -> (($mux | $logic_not) |* || $eq |+) -> $pmux -> @MUX >*]] ;				Encode/Decode Element 1 (\CO -> $eq) |+ -> @MUX ;				Encode/Decode Element 2 (\CO - > $eq) |+ -> $reduce_or -> @MUX ;				t[s]
					S	P	t(S)[s]	t(P)[s]	S	P	t(S)[s]	t(P)[s]	S	P	t(S)[s]	t(P)[s]	S	P	t(S)[s]	t(P)[s]	S	P	t(S)[s]	t(P)[s]	
42	rc4-prbs	crypto	3454	8437	5	2	2.206	1.289	8	3	2.335	2.165	1	1	0.839	0.584	Timeout				Timeout				9.42
43	reed_solomon_codec_generator	ecc	19784	40997	13	12	4.365	42.535	13	12	4.182	42.451	0	0	2.469	0.469	32	32	23.981	124.808	11	1	2.294	12.446	260.0
44	reed_solomon_decoder	ecc	3067	7038	29	15	4.962	9.704	29	15	**4.893**	9.957	4	4	2.806	2.89	551	551	42.926	457.906	76	76	12.0	43.912	591.96
45	risc16f84	proc	550	1047	3	2	0.503	0.21	4	3	0.59	0.329	0	0	0.216	0.007	**65**	**65**	3.648	22.195	0	0	0.008	0.008	27.71
46	rs_decoder_31_19_6	ecc	3124	6115	4	4	0.697	2.273	4	4	0.594	2.358	4	4	0.789	2.575	14	14	1.231	7.284	21	16	2.278	9.005	29.08
47	sasc	comm	188	298	**6**	**6**	0.504	0.143	**6**	**6**	0.614	0.161	1	1	0.221	0.045	2	2	0.214	0.08	1	1	0.171	0.043	2.2
48	scct	other	714	1526	3	2	0.336	0.228	3	2	0.332	0.231	0	0	0.112	0.012	41	41	2.636	4.1	0	0	0.013	0.015	8.02
49	sdr_ctrl	mem	1446	2440	6	6	1.334	1.43	7	7	1.501	1.72	7	7	1.343	1.966	115	115	7.549	71.918	38	31	3.468	5.853	98.08
50	sdspi	comm	871	1645	16	7	1.557	0.992	16	7	1.51	0.993	0	0	0.206	0.011	77	77	4.87	9.666	0	0	0.022	0.013	19.84
51	sha3	crypto	321	605	0	0	0.038	0.007	0	0	0.032	0.006	0	0	0.033	0.008	1	1	0.131	0.202	0	0	0.033	0.01	0.5
52	simple_gpio	other	42	80	0	0	0.001	0.001	0	0	0.001	0.001	0	0	0.001	0.001	0	0	0.001	0.001	0	0	0.001	0.001	0.01
53	simple_pic	other	105	212	0	0	0.002	0.002	0	0	0.003	0.004	0	0	0.003	0.004	3	3	0.243	0.052	0	0	0.003	0.003	0.32
54	simple_spi	comm	236	414	4	4	0.338	0.117	4	4	0.321	0.118	1	1	0.206	0.039	8	8	0.649	0.25	0	0	0.015	0.004	2.06
55	spimaster	comm	1323	2371	19	14	2.145	3.051	19	14	2.207	2.742	5	5	2.789	1.618	91	91	**7.854**	16.412	54	38	5.651	6.488	50.96
56	spislave	comm	28	58	0	0	0.011	0.001	0	0	0.014	0.001	0	0	0.015	0.001	3	3	0.297	0.024	0	0	0.014	0.001	0.38
57	ss_pcm	comm	77	126	1	1	0.128	0.028	1	1	0.159	0.027	0	0	0.016	0.002	0	0	0.015	0.003	0	0	0.015	0.003	0.4
58	statled	other	42	61	2	2	0.169	0.02	2	2	0.165	0.02	0	0	0.002	0.001	6	6	0.411	0.057	0	0	0.002	0.002	0.85
59	synchronous_reset_fifo	mem	49	64	2	2	0.203	0.028	2	2	0.191	0.029	0	0	0.003	0.002	1	1	0.108	0.015	0	0	0.003	0.001	0.58
60	systemc_rng	other	54	101	0	0	0.002	0.001	0	0	0.002	0.002	0	0	0.002	0.001	0	0	0.002	0.001	0	0	0.003	0.002	0.02
61	systemcaes	crypto	636	1252	8	3	1.158	0.841	8	3	1.169	0.753	3	3	0.593	0.682	31	31	3.04	12.213	4	4	0.826	0.996	22.27
62	usb	comm	3045	5689	17	17	2.489	8.98	21	21	2.774	11.277	**6**	**6**	**2.192**	3.464	89	89	8.317	53.121	77	58	12.307	36.056	140.98
63	usb_host_core	comm	1634	3223	11	11	1.114	2.393	11	11	1.166	2.421	1	1	0.665	0.33	169	169	12.751	61.301	2	2	0.272	0.589	83.0
64	usb_phy	comm	472	782	5	5	0.542	0.262	5	5	0.472	0.385	3	3	0.654	0.168	10	10	0.977	0.522	8	6	0.823	0.327	5.13
65	vga_lcd	video	992	1805	6	6	0.544	0.813	6	6	0.497	0.822	3	3	0.705	0.449	47	47	3.185	6.47	11	10	0.853	1.338	15.68
66	wb_conmax	soc	14385	25175	0	0	0.161	0.185	0	0	0.16	0.178	64	64	**69.37**	153.589	Timeout				0	0	0.15	0.185	223.98
67	wb_dma	soc	1245	2123	0	0	0.035	0.018	2	2	0.209	0.442	1	1	0.369	0.349	15	15	1.794	3.817	26	21	8.571	6.097	21.7
68	wb_flash	mem	53	65	2	1	0.182	0.016	2	1	0.185	0.017	0	0	0.002	0.001	7	7	0.444	0.065	0	0	0.002	0.001	0.92
69	wb_lpc	comm	36	59	0	0	0.011	0.001	0	0	0.016	0.001	1	1	0.204	0.017	3	3	0.305	0.035	1	1	0.153	0.015	0.76
70	wb_size_bridge	mem	304	607	0	0	0.004	0.004	0	0	0.007	0.007	2	2	0.227	0.094	9	9	0.672	0.528	12	5	1.042	0.256	2.84
71	wbif_68k	other	45	57	0	0	0.002	0.001	0	0	0.002	0.001	0	0	0.001	0.001	0	0	0.001	0.001	0	0	0.001	0.001	0.01
72	wbscope	test	108	174	3	3	0.237	0.082	3	3	0.22	0.058	0	0	0.003	0.003	0	0	0.005	0.002	0	0	0.004	0.002	0.62
73	xge_ll_mac	comm	14911	30001	1	1	0.577	3.524	1	1	0.705	3.353	3	3	0.896	8.809	44	44	4.862	174.041	6	5	1.001	13.355	211.12
74	xtea	crypto	152	302	1	1	0.113	0.055	**10**	**4**	0.648	0.165	1	1	0.207	0.036	14	14	1.022	0.604	7	5	0.808	0.162	3.82



(a) Overall architecture of the USB core



(b) Architecture of the PL block

Figure 6.1: Architecture of the *usb* core (line 62 in Table 6.4). (a) shows the overall architecture of the core, (b) shows the architecture of the protocol layer (PL) block. The numbers denote where and how many state machines we found in the modules of the architecture block. The images are taken from the documentation of the core,[4] the numbers are added.

6.2.1 State machines

We designed the *State Machine* PGSL pattern to match the functional kernel of synchronous state machines. A state machine contains a state register, that indicates in which of the possible states the state machine is for a given point in time. In each state, the next state value is calculated depending on the current state and conditions of other controlling signals. On a change of the clocking signal, the next state value is assigned to the state register, triggering a possible state change. The functional kernel, which we are matching, are the comparison containing the state register followed by a parallel multiplexer and an optional multiplexer chain. The output of the parallel multiplexer and optional multiplexer chain is the next state, which is fed back to the state register.

Illustrative example - USB 2.0 Function Core (usb)[3]

The *usb* design (line 62 in Table 6.4) is a suitable design to visualize how finding state machines can be used for design understanding as it offers extensive documentation to which we can relate our search results. The design is an Universal serial bus (USB) 2.0 compliant core with a Wishbone system on chip (SoC) interface and a physical layer (PHY) interface driven by the USB transceiver macrocell interface (UTMI). The *protocol layer (PL)* block contains packet assembly, disassembly, the protocol engine and a direct memory access (DMA) and memory interface. We find six distinct state machines in the design, performing the search on the individual modules allows us to pin point in which modules the state machines are implemented. Figure 6.1a shows the architecture of the core, Figure 6.1b shows the *PL* block of the core. We add annotations for the number of state machines found in the modules of each architecture block.

When matching against the *State machine* pattern, we find state machines controlling the steps to assemble and disassemble packets in the *PL* block. Each solution is a subgraph of the *usb* core's design graph, as an example we can examine the solution for the package assembly state machine (see Figure 6.2). The data flip-flop cell $dff is the state register, the following cells in parallel are comparisons involving the state register value. The parallel multiplexer $pmux cell takes the results of the comparisons to form with the following multiplexer $mux cell the next state, which in turn is an input to the state register.

[3] *OpenCores project "USB 2.0 Function Core"*. https://opencores.org/projects/usb. Accessed: 2019-11-26.
[4] *OpenCores project "USB 2.0 Function Core"*. https://opencores.org/projects/usb. Accessed: 2019-11-26.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

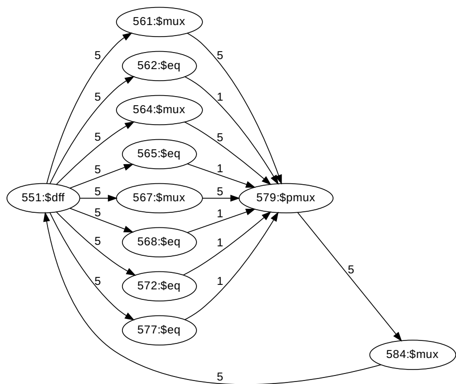


Figure 6.2: Solution as subgraph of the design graph for state machine in the packet assembly module of the *usb* design.

Listing 6.2: Simplified version of the state machine in the package assembler module of the *usb* design (rtl/verilog/usbf_pa.v). Lines that do not affect the *next_state* are omitted, the full code is shown in Appendix A. This illustrates a typical state machine structure that we find using the *state machine* PGSL pattern.

```
1  next_state = state;     // Default don't change current state
2  case(state)
3          IDLE: begin
4                  if(send_zero_length_r && send_data) begin
5                          next_state = WAIT;
6                  end
7                  else if(send_data) begin
8                          next_state = DATA;
9                  end
10         DATA: begin
11                 if(!send_data && tx_ready && tx_valid_r) begin
12                         next_state = CRC1;
13                 end
14         end
15         WAIT: begin
16                 next_state = CRC1;
17         end
18         CRC1: begin
19                 if(tx_ready) begin
20                         next_state = CRC2;
21                 end
22         end
23         CRC2: begin
24                 if(tx_ready) begin
25                         next_state = IDLE;
26                 end
27         end
28 endcase
```

The solution graph in Figure 6.2 is the functional kernel of the state machine's HDL description code, a simplified version is shown in Listing 6.2. This particular state machine progresses through its states *IDLE*, *DATA*, *WAIT*, *CRC1*, and *CRC2* depending on the signals *send_zero_length*, *send_data*, *tx_ready* and *tx_valid_r*.

Different results for search and post-processing

The designs *ca_prng* (line 11 in Table 6.4) and *nextz80* (line 36 in Table 6.4) are of special interest, because they both show different numbers of results before and after post-processing. For the *ca_prng* post-process reduces from 32 search solutions to 1, for *nextz80* the reduction is from 5 to 1. This indicates that the search operation shows multiple state machines, which in fact are just one. Both cases can be explained by analyzing the solution graphs the search operation produces and the module the post-processing step produces.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

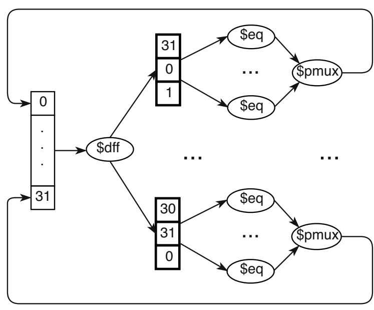


Figure 6.3: Parallel state machine structure found in the *ca_prng* design. Each of the 32 parallel state machine takes three bits of the $dff register cell as state value and calculates one new bit for the register. Post-processing detects this as one state machine.

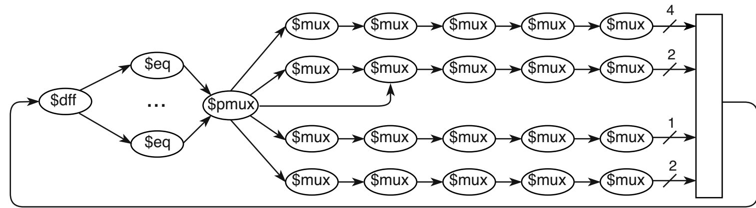


Figure 6.4: State machine structure with different feedback paths found in the *nextz80* design. Each of the five different feedback $mux cell chain paths is a distinct result of the search operation. Post-processing detects this as one state machine.

The *ca_prng* design, a cellular automata pseudo random number generator (PRNG) has a two-level hierarchy state machine, where the level-two state machines are in parallel. Each of the level-two state machines interprets a three-bit subvector of the level-one state as its own state, and modifies one bit of the level-one state machine. This structure is sketched in Figure 6.3. For example, the first search result takes the bits 31, 0 and 1 as state variable to calculate the new bit 0 of the register. The following 31 parallel state machines each take different three-bit subvectors of the $dff state register to calculate the next values for bit 1 to 31. A design graph has no cell ports, they are abstracted away to simplify the search space with our design graph abstraction. Therefore, all solutions contain the $dff node. Post-processing merges the 32 state machines, to one large state machine.

In the *nextz80* design the state machine has multiple feedback paths from the $pmux cell to the state register $dff. The search operation interprets each different feedback path as an individual state machine. Post-processing detects, that it is in fact only one state machine. The issue is sketched in Figure 6.4.

Performance analysis

Searches with the *State Machine* pattern show good performance, all searches on the designs except one take under 10 seconds. To compare performance on different designs, we study the statistics, which we defined in Table 6.1. The state machine search for the medium-sized design *usb* (see line 62 in Table 6.4) takes 2.2 seconds, while the biggest design in which we found state machines (*wb_conmax*, line 66 in Table 6.4) takes 153.6 seconds. Compared to the *usb* design, the design *wb_conmax* is significantly bigger with similar complexity: it has 14385 cells (*usb*: 3045) and 25175 connections (*usb*: 5689), and a similar average degree of 3.5 (*usb*: 3.7). Comparing the number of cells and remaining connections after filtering the designs to the node types in the *state machine* pattern gives an overview of the resulting search space size (see Table 6.5). Filtering the design to the cell types contained in the *State Machine* PGSL pattern significantly reduces

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Table 6.5: Comparing the search space for the designs *wb_conmax* and *usb*. To save space we abbreviate the design *wb_conmax* as *wb*. Highlighted are the number of cells, for cell types, that are matched in the dominant subpattern (($mux | $logic_not) |* || $eq |+ .

Cells	usb	usb filtered	wb	wb filtered
$dff, $adff	329	329	371	371
$eq	178	**178**	1128	**1128**
$logic_not	95	**95**	243	**243**
$pmux	74	74	209	209
$mux	644	**644**	5094	**5094**
others	1725	0	7340	0
total	3045	1320	14385	7045

Statistic	wb	wb filtered	usb	usb filtered
Connections	3045	1882	25175	10337
Average degree	3.7	3	3.5	2.9
Max degree	249	43	371	35

the complexity of the search space as can be seen with the number of total cells and number of connections.

Analyzing the pattern and the two different search space characteristics combined with knowledge of the search algorithm, we can explain the much higher search time for the *wb_conmax* design. The most time consuming search operations in the *State Machine* pattern are the search for the parallel quantified subpattern (($mux | $logic_not) |* || $eq |+ and the multiplexer chains @MUX >* . The subpattern which dominates the workload of the search operation depends on the design structure. Neither the less, designs which contain more nodes of types that are contained in the two quantified subpatterns will, as a rule of thumb, have longer search times. The *wb_conmax* design contains significantly more nodes that are contained in these subpatterns than the *usb* design, this leads to a higher workload in the solving process of these subpatterns.

6.2.2 Counters

To detect counters in a given HDL design, we designed the PGSL patterns *Counter 1* and *Counter 2*. *Counter 1* is aimed at finding counter structures, that are incremented by constant values. Therefore, it contains a \CO node connected to the $add addition node. The PGSL pattern *Counter 2* contains no \CO node and therefore also matches counter structures that are not incremented by a constant value, e.g., the value for incrementation is calculated and stored in a register.

Illustrative example - simple asynchronous serial controller (sasc)[5]

A simple example to visualize results for the *Counter 1* pattern is the *sasc* design (line 47 in Table 6.4). This design is a simple asynchronous serial controller, to receive and transmit via the universal asynchronous receiver transmitter (UART) protocol. Communication via UART is bidirectional with a one-bit line for receiving and a one-bit line for transmitting. The data is transferred at a given data rate (baud rate). A sketch of the architecture of the *sasc* core can be seen in Figure 6.5. The data interface of the *sasc* core offers communication with eight-bit words. As the UART receive and transmit is one bit at a time, the incoming and outgoing data words need to be buffered. This is done with FIFO units. Each of these units contains multiple eight-bit memory cells and a read-and-write pointer to keep track, in which of the memory cells the next data word should be written and from which memory cell the next data word should be read. These pointers are incremented on read-and-write operations. These incrementation operations implement

[5] *OpenCores project "Simple Asynchronous Serial Controller"*. https://opencores.org/projects/sasc. Accessed: 2019-11-26.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

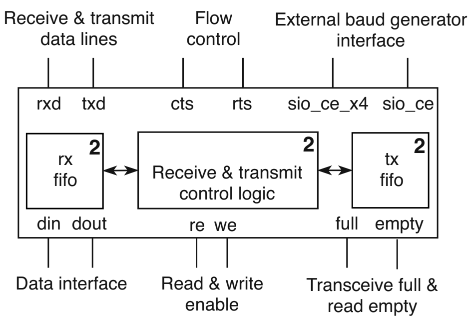


Figure 6.5: Architectural view of a simple asynchronous serial controller (line 47 in Table 6.4). The top design instantiates one FIFO for receiving data and one FIFO for transmitting data. Each of the FIFO contains two counters. Additionally, the top design contains two counters to count the position of the transmitted and received bit. The bold numbers indicate how many counters we found in each of the architecture blocks.

Listing 6.3: *Verilog* code for the write pointer from the *sasc* design (rtl/verilog/sasc_fifo4.v starting at line 90). The write-pointer *wp* signifies the position in the FIFO, where the next data word will be written to. This write-pointer implements a counter, that counts up if the write-enable *we* is true.

```
1 always @(posedge clk or negedge rst)
2     if(!rst)
3         wp <= 2'h0;
4     else if(clr)
5         wp <= 2'h0;
6     else if(we)
7         wp <= wp_p1;
8
9 assign wp_p1 = wp + 2'h1;
```

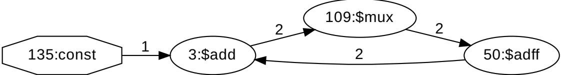


Figure 6.6: Solution as subgraph of the design graph for the write pointer counter in the FIFO of the *sasc* design.

counters. Additionally, the receive-and-transmit logic contains a counter to count the position of the transmitted and received bit. Listing 6.3 shows how such a simple counter, e.g. the write pointer counter of one of the FIFO units can be modeled with HDL *Verilog*. In a process that is triggered by the positive edge of the signal *clk* (line 1) the signal *wp_p1*, which is always equal to *wp* + 1 (line 9), is assigned to the write pointer *wp* (line 7) if the write enable signal *we* is set (line 6). The corresponding solution graph is shown in Figure 6.6. The multiplexer node between the $add node and the $adff register node $mux multiplexes if the write enable signal *we* is set and therefore the new value of *wp* should be *wp_p1* or if the old value of *wp* should be kept.

Different results for search and post-processing

With the *Counter 1* and *Counter 2* PGSL patterns the post-processing step often has to merge solutions from the search operation into one counter. We see this in the result overview table Table 6.4 when analyzing at the many different numbers comparing search results (S) and post-processing results (P) in the *Counter 1* and *Counter 2* columns. An example is the I2C slave IP core *i2cslave* (line 23 in Table 6.4). For the pattern *Counter 1* and *Counter 2* we each find six matches with the search operation and post-processing produces only two results. This result tells us that (1) there are only counters incremented by constants found, and (2) post-processing eliminates/merges duplicates. We can state (1)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

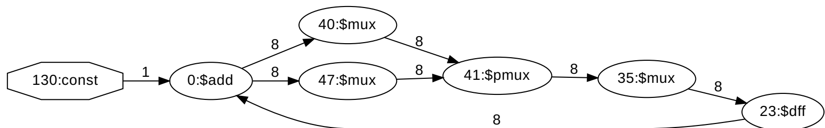


Figure 6.7: Counter structure from the *i2cslave* design. The search operation finds two counters, as two multiplexer-chain paths connect the $add and $dff node. Post-processing merges this to one counter.

because *Counter 2* (that also matches non-constant incrementing counters) is a subpattern of *Counter 1* (that only matches constant incrementing counters). Statement (2) requires closer inspection. Analyzing the solution graphs, we find out that in fact only two counters exist in the design. The reason are multiplexer structures that are not simple chains. For example, the first counter returned by the post-processing contains such a multiplexer structure (see Figure 6.7). The search operation finds two chains to connect the $add and $dff nodes (one via node 40 and one via node 47) and therefore interprets them as two distinct solutions. Post-processing merges them into one counter. These kind of multiplexer structures are the dominant reason for post-processing to merge search results of the counter patterns in the designs used for our experiments.

Another case where the search operation finds multiple counters which can be merged to one, can be found in the *dpll-isdn* design (line 13 in Table 6.4). One of the counters found in this design has two modes. In mode one the register is incremented by one; in the second mode the register is incremented by two. The search operation interprets this as two distinct counters, post processing merges them into one.

Non-constant incrementing counters

Searching for counters with the two different PGSL patterns allows us to easily spot if a design contains counters that are not incremented by constant values. If the number of results for the pattern of *Counter 2* is larger than for *Counter 1*, the design contains non-constant incrementing counters. Examples are given below:

- *ima_adpcm_enc_dec* (line 24 in Table 6.4): In this design we find two non-constant incrementing counters. The first counter is incremented by a primary input of the design, the second is incremented by the value of a preceding addition operation.
- *xtea* (line 74 in Table 6.4): One of the counters found, is incremented by a delta value, that is stored in a register.

Performance analysis

Searches for the *Counter 1* and *Counter 2* pattern show good performance, all searches on the designs except one take under 10 seconds. The search time difference between searches for pattern *Counter 1* and *Counter 2* differ slightly. Searches with *Counter 1* impose an additional constraint to the search. The filtering during search can eliminate $add nodes, that do not have an incoming connection from a \CO node. Nevertheless, this has no significant impact on the search time difference. The maximal difference we found is of a factor two. For the design *ftdi_wb_bridge* (line 19 in Table 6.4) the search for *Counter 1* takes 1.6 seconds, while the search for *Counter 2* takes 0.9 seconds. For the design

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Table 6.6: Comparing the search space and performance for the designs *reed_solomon_decoder* and *m32632*.

Cells	reed_solomon_decoder	m32632
$add	105	59
$dff, $adff, $dlatch	466	399
$pmux, $mux	1211	955
total	1762	1453

Statistic	reed_solomon_decoder	m32632
Connections	3172	2228
Average degree	3.6	3.3
Max degree	58	70
Search time	**4.9 s**	**78.2 s**
Results @MUX >*	**2648**	**141273**

jt51 (line 25 in Table 6.4) we find the opposite relation: the search for pattern *Counter 1* takes 2.3 seconds, the search for *Counter 2* takes 4.5 seconds.

The dominant subpattern search for the counter patterns regarding the search time is the search for multiplexer feedback chains which are represented by the subpattern @MUX >*. As we can rule out the influence of the \CO node, we examine at the *Counter 2* pattern to investigate the impact of the dominant subpattern. As an example, we found the two designs *reed_solomon_decoder* and *m32632*. Although after filtering to the cells in the *Counter 2* pattern they have a similar size, cell count distribution and complexity, they show radically different search times (4.8 s vs 78.2 s). Therefore, not only the number of multiplexer cells in the design is a decisive factor for the search time, but also how these multiplexers are connected. An overview of the designs' cell distribution and statistics are summarized in Table 6.6. To judge the connectivity of the multiplexers we conducted searches on both design using the subpattern @MUX >*. This metric finally shows that the *m32632* design has a multiplexer connectivity that is much higher than the connectivity found in the *reed_solomon_decoder* design. A search on the design *m32632* has to handle 141273 multiplexer chains, while the search on the design *reed_solomon_decoder* only has to work with 2648 chains (factor 53 less!). Therefore, we conclude that the multiplexer connectivity of a design is the dominating factor regarding search time for the counter patterns.

6.2.3 Encode/decode elements

We designed two *Encode/Decode Element* PGSL patterns to detect structures in a design where values are compared in parallel and fed to a multiplexer that determines how to decode or encode based on the value. The experiment shows that these two patterns indeed match to encoders, decoders, and lookup tables. The pattern *Encode/Decode Element 1* matches to structures where one value is compared to multiple constants in parallel, and each of the comparisons imply a different behavior for the following circuit. In contrast, pattern *Encode/Decode Element 2* with the additional $reduce_or can match to comparisons that check if a given value is in a set of values.

Illustrative example - single-cycle logarithm function (fast_log)[6]

A simple example to visualize results for the *Encode/Decode Element 1* and *Encode/Decode Element 2* patterns is the *fast_log* design (line 16 in Table 6.4). This single-cycle fast logarithm calculation core operates with three stages: a priority encoder followed by a barrel-shifter and a lookup table (LUT). For this design after post-processing we find two solutions for *Encode/Decode Element 1* and one for *Encode/Decode Element 2*. The search and post-processing stage extract the priority encoder (one solution from *Encode/Decode Element 1*) and the output LUT (the single post-processing

[6] *OpenCores project "Logarithm function, base-2, single-cycle".* https://opencores.org/projects/fast_log. Accessed: 2019-11-26.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Listing 6.4: Part of the lookup table code found in the design *fast_log* (Log2flowthru.v). The comparisons and resulting LUT output in line 2, 5, and 8 are matched by the pattern *Encode/Decode Element 1*, the rest is matched by *Encode/Decode Element 2*.

```
case (barrelout)
        0:      LUTout  =       0;
        1:      LUTout  =       1;
        2:      LUTout  =       1;
        3:      LUTout  =       2;
        4:      LUTout  =       3;
        5:      LUTout  =       3;
        6:      LUTout  =       4;
        7:      LUTout  =       5;
        8:      LUTout  =       5;
...
end case
```

result for *Encode/Decode Element 2* and one solution from *Encode/Decode Element 1*). The code for the output LUT (see Listing 6.4) gives an example why we need two slightly different patterns. The comparisons in line 2, 5, and 8 each give a unique output value to the signal *LUTout* and therefore match to *Encode/Decode Element 1*. In contrast, the comparisons in line 3 and 4 produce the same output. They are connected in the synthesized design by a $reduce_or cell and subsequently match to *Encode/Decode Element 2*. Without searching for the pattern *Encode/Decode Element 2* we would not find all elements of the LUT.

Instruction and address decoders

Beside LUTs the Encode/Decode Element patterns also find address and instruction decoders. Examples are given below:

- *mips_16* (line 33 in Table 6.4): In this processor core we find the decoder that decides whether a given opcode is a branch or a store instruction. Additionally, we find the address decoder that handles access to special registers for the WISHBONE SoC communication interface.
- *risc16f84* (line 45 in Table 6.4): In this processor we find the instruction decoder and the address decoder that is in charge of handling accesses to memory mapped periphery.

Different results for search and post-processing

For most search operations using the pattern *Encode/Decode Element 2*, post-processing merges search results to bigger comparison trees, where each branch that is a distinct search result connects to a common multiplexer. For example, the design *fast_log* contains 13 search results with the $pmux node as common element, that can be merged to one solution. A resulting merge for two of the search solutions is shown in Figure 6.8.

Reasons for timeouts

The two decode-element patterns are the only patterns for which our time limitation of 20 minutes for search + post-processing was violated for certain designs. We marked these cases in the results overview table (Table 6.4) as "Timeout". These time limitation violations have two distinct reasons: (1) timeout in the search operation, due to large search space, and (2) timeout in the post-processing stage, due to high workload from many matches from the search operation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

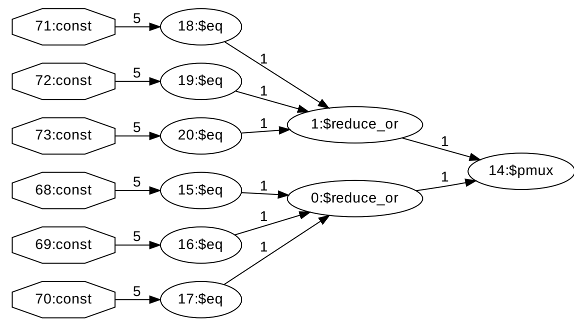


Figure 6.8: Branch merging performed by the post-processing stage in the design *fast_log*. Each of the two branches connected to the $pmux multiplexer are a distinct result from the search operation. Post-processing can merge them to comparison tree.

Table 6.7: Comparing the search space and performance for the designs *spimaster*, *m32632*, and *jt51*. Rising search time correlates with rising number of $eq and \CO cells and a higher number of connections.

Cells	spimaster	m32632	jt51
$eq	**124**	**1076**	**1786**
\CO	**534**	**2840**	**2136**
$mux, $pmux	389	955	506
total	1047	4911	4428

Statistic	spimaster	m32632	jt51
Connections	**1022**	**3605**	**13437**
Average degree	2	2.3	1.7
Max degree	92	412	513
Search time	**7.8 s**	**20.8 s**	**33.3 s**

The designs *aes-128_pipelined_encryption* (line 4 in Table 6.4) and *aes-encryption* (line 5 in Table 6.4) both fall into category (1). Both designs are very large, with over 100000 cells, and contain a large number of cells, that have to be checked during the search process to match the patterns. The design *aes-128_pipelined_encryption* contains 51000 $eq and 51642 \CO cells after filtering, the numbers for the design *aes-encryption* are similarly large. The subpattern ($eq ->\CO)|+, that matches to these cells is a key part of the *Encode/Decode Element* pattern. The CSP solver of our search operation has to check which pairs of $eq and \CO cells are connected. With such large numbers of cells the number of permutations to try is to large for the search operation to terminate in the given time boundary.

For the designs *wb_conmax* (line 66 in Table 6.4) and *rc4-prbs* (line 42 in Table 6.4) the search operations terminate in under 30 seconds, but the post-processing violates the time boundary. For both designs the search operation finds over 700 matches. Post-processing has to extract the input cone for each match and subsequently compare the resulting modules for duplicates. This high workload does not terminate in the given time boundary.

Performance analysis

Searches with the *Encode/Decode Element* patterns show the largest variance and highest values concerning search times. As mentioned for the reasons of timeouts, in this pattern the subpattern (\CO -> $eq) |+ is the dominant factor for search times, specifically the number of \CO and $eq cells a design contains and the connectivity of the design. The CSP solver of our search operation has to check which pairs of $eq and \CO cells are connected. The more $eq to \CO pairs have to be checked and found, the more search time is spent for this subpattern. We visualize this by showing three designs for the pattern *Encode/Decode Element 1* with rising search times in Table 6.7.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 7

Discussion

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The experimental results indicate that the search method presented in this thesis is a solution to the research questions. Our methodology is effective and efficient for finding structural patterns in HDL designs. PGSL offers an intuitive approach to model structural kernels of high-level functional primitives such as counters, state machines and decoders/encoders. Our search algorithm can handle searches with PGSL patterns that have built-in structural variability (quantifications). We demonstrate the efficiency and effectiveness with a large-scale experiment on 74 real-world designs from *OpenCores*.[1] The experiments cover a wide range of different sizes: the smallest design has 29 cells with 48 connections (design *random_pulse_generator*; line 41 in Table 6.4), the largest design has 104785 cells with 158657 connections (design *aes-128_pipelined_encryption*; line 4 in Table 6.4). In this experiment, we were able to find the structural kernels of the high-level functional primitives state machines, counters and decoders/encoders in reasonable times. 214 of 370 (58%) search and post-process operations finish in under 1 second, 307 of 370 (82%) in under 10 seconds (see Table 7.1). The average search time over all designs and patterns is 2.8 seconds, the maximal search time is 141.5 seconds.

In the following subsections we highlight and discuss aspects of our experimental results.

Table 7.1: Duration statistics for the pattern-search and post-processing operations (experiment with 74 designs, 5 patterns, description in Chapter 6). The percentages are rounded.

Duration	Search	Search + post-processing
Under 1s	254 (69%)	214 (58%)
1 s to 10s	94 (25%)	93 (25%)
10 s to 30s	11 (3%)	30 (8%)
Over 10 s	9 (2%)	28 (8%)
Timeout (>20 min)	2 (0.5%)	5 (1%)

Quantification

A characteristic feature concerning pattern modeling in our pattern-search methodology are quantified subpatterns. With PGSL we can take any subpart of a pattern and quantify it serially or in parallel. We use serial quantifications in our patterns for variable-length serial feedback paths (multiplexer chains) in counters (pattern *Counter 1* and *Counter 2*) and state machines (pattern *State Machine*). These feedback paths are the control path for the next state (for state

[1] *OpenCores*. https://opencores.org/.

machines) or the next counter value (for counters). The length of feedback multiplexer chains can vary, as different state machines are controlled by different control signals and different counters have to take into account different conditions for when to increment or reset a counter (e.g., clear, reset signals). Without serial quantification, a user of our search method would have to specify multiple patterns for different lengths of multiplexer feedback chains. The experimental results confirm that serial quantification is an important feature. In the design *risc16f84* (line 45 in Table 6.4) we find the biggest multiplexer-chain for a counter with 15 multiplexer cells in serial. Without quantification, we would need 16 patterns for counters with multiplexer-chain length 0 (no multiplexer in feedback path) up to length 15 to find this counter. Similarly, we can see the utility of parallel quantification with parallel comparisons in decoders/encoders (pattern *Decode/Encode Element 1* and *Decode/Encode Element 2*) and state machines (pattern *State Machine*). Again, the experimental results confirm the importance of the parallel quantification feature. In the design *jt51* (line 25 in Table 6.4) one of the solutions has 511 parallel comparisons connected to a common multiplexer.

Impact of post-processing

We use post-processing to demonstrate that the functional kernels we extract can be processed to full versions of functional primitives and to merge search results that matched to the same instance of a functional primitive. The post-processing extracts modules from the search results including the input cone of all the cells included in the search result into *Verilog* modules. The post-processed results contain the control signals that influence the functional primitive. Post-processing extracts the input cone for each match and subsequently compare the resulting modules for duplicates. This poses a high workload for the post-processing. For almost 47% of the search and post-processing operations that return at least one search result, the runtime for post-processing is higher than the runtime for the search operation. In the worst case for the design *qspiflash* (line 40 in Table 6.4), with 661 search results and no reduction by post-processing the runtime of post-processing is a factor 46 higher than the runtime for the search operation (search: 43s, post-processing: 928s). Nevertheless, post-processing is a procedure that has an impact on many of the search results in our experiment. In 26% of all searches that have at least one search result, post-processing was able to merge search results. A detailed view can be seen in Table 7.2. One example where the post-processing is very successful is for merging the control paths of counter search-results. In our counter patterns we match the feedback path of a counter as multiplexer-chain. The feedback path is the control path of the counter and can be formed by multiple different multiplexer chains, as the feedback path is not a chain but rather a tree. For such a case, the search operation returns multiple distinct results, post-processing creates a module for each of them, discovers the duplicate and only outputs one module that represents merged version. We present other examples where post-processing successfully merges search results in Section 6.2.

Table 7.2: Overall impact of post-processing. For every pattern we show the number of designs that have search results for this pattern, and the number of those searches that were impacted by post processing. Post-processing had an impact if the number of results after post processing was smaller due to merging of search results.

Designs ..	*Counter 1*	*Counter 2*	*State Machine*	*Encode/Decode Element 1*	*Encode/Decode Element 2*
with search results	46	48	35	58	38
impacted by post-processing	14 (30%)	17 (35%)	2 (6%)	0 (0%)	25 (66%)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Timeouts

We set a time limitation of 20 minutes for the search and post-processing. Searches and post-processing that violate this timeout are marked as "Timeout" in Table 6.4. We find timeouts for 4 designs and from search and post-processing with the *Encode/Decode Element 1* and *Encode/Decode Element 2* patterns. These time limitation violations have two distinct reasons: (1) timeout in the search operation due to large search space, and (2) timeout in the post-processing stage due to high workload from many matches from the search operation. Although we set the timeout to an arbitrary value, the occurrence of timeouts shows that our search and post-processing has limitations.

An example for category (1), timeout due to large search space, is the search in design *aes-128_pipelined_encryption* (line 4 inTable 6.4). The critical subpattern that leads to timeout is "($eq -> \CO)|+". The design contains 51000 $eq and 51642 \CO cells after filtering. In order to resolve this subpattern, the constraint satisfaction problem (CSP) solver of our search operation has to check which pairs of $eq and \CO cells are connected. With such large numbers of cells the number of possible permutations is too large for the search operation to terminate in the given time boundary.

An example for category (2), high workload for post-processing due to many matches, is the design *rc4-prbs* (line 66 in Table 6.4). Although the search operations terminate in under 30 seconds with over 700 search results, the workload for post-processing is too high. Post-processing has to extract the input cone for each match and subsequently compare the resulting modules for duplicates.

Performance factors

Our pattern search algorithm solves a subgraph-isomorphism problem between the pattern graph and the design graph of a given synthesized HDL design using hierarchical matching and a custom CSP solver for combining matches of subpatterns. Its performance in general is dependent on the design graph after filtering to the node types contained in the pattern ("filtered design graph") and the pattern which is searched for. Our initial filtering is an efficient measure to reduce the search space. For all patterns we reduce the number of nodes in a design on average by at least 35% and the number of connections on average by at least 59%. A closer inspection of the impact of filtering separated by the five patterns we use can be seen in Table 7.3.

Solving a constraint satisfaction problem (CSP), the method we choose to hierarchically combine matches of subpatterns, is an NP-complete algorithm which has exponential worst case behavior. Although we did not conduct a formal run-time-complexity analysis (out of scope for this thesis, possible future work), we expect that our search method overall exhibits an exponential worst-case behavior inherited from the CSP. This assumption is underlined by the fact that we use simple backtracking search without constraint propagation, variable ordering, and value ordering. Therefore, increasing the number of nodes in a design graph will have an exponential impact on the search time.

Overall, performance estimation is complicated by the fact that we do not search matches for one uniform graph, but rather pattern graphs that are nested and contain quantified subpatterns. This leads to interesting side effects and sub-patterns that can dominate the search. For example, we observe that high connectivity of elements that are matched in serial-quantified subpatterns can have a high impact on search times. If these elements are highly connected in the design graph, the subpattern search results in more matches for this quantified subpattern. This explosion of subpattern

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Table 7.3: Average decrease of the search space parameters number of nodes and number of connections after filtering to types contained in a given PGSL pattern. The decrease percentages are averaged for the searches in the 74 designs and separated for each of the five patterns we use in the experimental chapter (see Chapter 6).

Average decrease of ...	*Counter 1*	*Counter 2*	*State Machine*	*Encode/Decode Element 1*	*Encode/Decode Element 2*
Number of nodes	41.5%	70.4%	59.3%	37.5%	35.6%
Number of connections	66.6%	76.2%	62.9%	66.6%	59.9%

Table 7.4: Impact of high connectivity for quantified subpattern searches. We compare the search space and performance for searches with pattern *Counter 2* for the designs *reed_solomon_decoder* and *m32632*.

Cells	reed_solomon_decoder	m32632
$add	105	59
$dff, $adff, $dlatch	466	399
$pmux, $mux	1211	955
total	1762	1453

Statistic	reed_solomon_decoder	m32632
Connections	3172	2228
Average degree	3.6	3.3
Max degree	58	70
Search time	**4.9 s**	**78.2 s**
Results @MUX >*	**2648**	**141273**

matches corresponds to a big domain which we have to iterate in the CSP solver when combing these subpattern candidates with candidates higher in the hierarchy. An example for this can be seen for the two designs *reed_solomon_decoder* and *m32632* with the *Counter 2* pattern. We compare their search space and search performance in Table 7.4. Although after filtering to the cells in the *Counter 2* pattern they have a similar size, cell count distribution and complexity, they show radically different search times (4.8 s vs 78.2 s). To determine the reason behind this discrepancy, we examine the subpattern searches. The pattern *Counter 2* contains a multiplexer chain subpattern @MUX >*. A search on the design *m32632* has to handle 141273 multiplexer chains, while the search on the design *reed_solomon_decoder* only has to work with 2648 chains (factor 53 less!).

Another general note on performance: whenever possible, pattern searches should be done on the non-flattened version of an HDL design. We investigate searches on flattened graphs in Chapter 6 in order to examine the performance of our search method on large design graphs. However, searching on non-flattened designs with modules divides the pattern search into multiple, most probably less intensive searches, as our search processes each module separately. Additionally, searching in non-flattened versions of a designs simplifies pinpointing correlations between search results and HDL code.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 8

Conclusion

In the framework of this thesis we addressed the problem of finding high-level functional primitives in *Verilog* hardware description language (HDL) designs by using structural pattern matching. Being capable of identifying functional primitives supports reverse- and verification engineers in checking a given design against its specification, identify errors, perform security evaluations, and deepen overall understanding of a design. Our work encompasses the modelling of appropriate search patterns for selected functional primitives, developing a search strategy to find these patterns in a given design. Furthermore, we demonstrated the efficiency and effectiveness of our chosen search strategy. We used the design graph abstraction (an abstraction of register transfer level (RTL) netlists; see Section 3.1) as simplified search space and the pattern graph specification language (PGSL) (see Chapter 4) to model patterns for the structural kernels of functional primitives. Our hierarchical solving approach (see Chapter 5) is based on the following key points: (1) initial and continuous filtering of the search space based on the chosen search pattern, and (2) a search-and-combine strategy using a custom constraint satisfaction problem (CSP) solver. Our pattern search is implemented as plugin to the open-source synthesis suite *Yosys*.[1] In Chapter 6 we evaluated our methodology with a large-scale experiment using 5 patterns we developed, and 74 *Verilog* designs from *OpenCores*.[2] We discussed the overall effectiveness and efficiency of our search methodology in Chapter 7.

In Chapter 2 we analyzed existing approaches for finding high-level structures in digital designs. We discussed their strengths and deficiencies, and contextualized our methodology that is able to face the problem of structural variability in subcircuits with similar functionality. Existing approaches check subcircuits of designs against reference designs either for functional equivalence, or structural equivalence, or use a mixed multi-step procedure based on functional- and structural equivalence. Functional approaches struggle to capture the characteristic functionality of circuits, while structural methods struggle to identify structural variability of components with similar functionality. Mixed approaches try to combine the strength and weaknesses of both approaches, but end up with multiple algorithms for different classes of components. In contrast, we developed a uniform methodology that checks for structural equivalence and faces the challenge of structural variabilities with quantified subpatterns.

[1] C. Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/.
[2] *OpenCores*. https://opencores.org/.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We demonstrated the effectiveness of our chosen pattern modelling language PGSL in Chapter 4 by developing patterns for the following functional primitives: counters, state machines and elements of decoders/encoders. Using PGSL patterns that represent graphs with quantified subpatterns, we were able to model the structural variances of different instances of a functional primitive. A quantified subpattern matches a certain number of times in serial or in parallel depending on the specified quantification. As an example, we quantified the feedback paths of counters (serial quantification) and the current state comparisons of state machines that are involved in calculating a next state (parallel quantification). For counters and elements of decoders/encoders we each developed two patterns to model different variants of these functional primitives. Together with the pattern for state machines we ended up with 5 patterns that we used in our experiments. The five patterns use the full range of characteristic features provided by PGSL.

As both our search space ("target graph") and the search patterns ("search graph") are graphs, we face a variant of the subgraph isomorphism problem. The main challenge concerning our variant of the problem were the implications of allowing quantification in a pattern. A pattern with quantified subpatterns (depending on the specified quantification) possibly represents an infinite number of graphs which have to be found as subgraphs in a target graph. Quantifications that allow zero occurrence of a subpattern pose an additional challenge. Matching for these cases changes the structure of our search graph: a serial quantification with zero occurrence places a short, a parallel quantification with zero occurrence places a split in the graph at the place of the subpattern. Therefore, we chose a search-and-combine approach with a custom CSP solver that fulfills our needs. Starting with subpatterns that match to one node in the target graph, we created candidates for each of the subpatterns at every hierarchy of our search pattern. Candidates are subgraphs of the target graph. Serial quantifications, parallel quantifications and combining candidates to matches of bigger subpatterns are solved with our custom CSP solver. To reduce the search space, we filtered the target graph prior to the search to contain only nodes of types that appear in the chosen pattern and also filtered the candidates of single nodes in our patterns by neighborhood profiling. As an end result of our pattern search process the user faces all the subgraphs of the target graph that match to the chosen pattern.

Our experiments verified the efficiency and effectiveness of our chosen search strategy. The experiments covered a wide range of different sizes and complexity and therefore give a good measure for real world performance: the smallest design has 29 cells with 48 connections, the biggest design (a whole advanced encryption standard (AES) encryption core) has 104785 cells with 158657 connections. We were able to find the functional kernels of the high-level elements state machine, counter and decoder/encoder in reasonable times: 214 of 370 (58%) search operations finish in under 1 second, 307 of 370 (82%) in under 10 seconds. The average search time over all designs and patterns is 2.8 seconds, the maximal search time is 141.5 seconds. The experimental results demonstrate that quantification is an effective technique to capture the structural variabilities of a functional primitive and furthermore is superior to modelling multiple structural similar patterns. For example, one search result for a decoder/encoder element matches 511 parallel instances of a subpattern. Our initial filtering is an efficient method to reduce the search space. For all patterns we reduced the number of nodes in a design on average by at least 35% and the number of connections on average by at least 59%. Search results can be used for further analysis of a design. As demonstration we used a post-processing step that extracts each search result as a *Verilog* module.

Future work can move in several directions. As already discussed in Chapter 7, allowing quantified subpatterns also leads to side effects such as subpattern searches that can dominate the search. For example, we observed that high connectivity of elements that are matched in serial-quantified subpatterns can have a high impact on search times. If

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

these elements are frequently connected in the design graph, the subpattern search results in more matches for this quantified subpattern. This explosion of subpattern matches corresponds to a high number of candidates which we have to iterate in the CSP solver when combing these subpattern candidates with candidates higher in the hierarchy. A common technique to face a high number of candidates in a CSP is to use value and variable ordering and constant propagation. Examples for such techniques are listed in Section 3.2. Therefore, we propose further research, which of the existing heuristics and constraint propagation techniques are the most appropriate to speed up our pattern searches. Additionally, as multithreading is common nowadays, subpattern searches could be divided into multiple threads to speed up the overall solving process. This of course implies profiling which subpatterns can be solved individually and synchronization of the candidate storage for efficient use of memory.

Although there is still room for improvement, we are convinced that the pattern search methodology presented in this thesis is a valuable tool that can aid reverse- and verification engineers in design understanding.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bibliothek Your knowledge hub
TU WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[1] P. Barceló, L. Libkin, and J. L. Reutter. "Querying Regular Graph Patterns". In: *J. ACM* 61.1 (Jan. 2014). ISSN: 0004-5411. DOI: 10.1145/2559905. URL: https://doi.org/10.1145/2559905.

[2] *Bison general-purpose parser generator*. https://www.gnu.org/software/bison/.

[3] F. Brglez and H. Fujiwara. "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran". In: *Proceedings of IEEE Int'l Symposium Circuits and Systems (ISCAS 85)*. IEEE Press, Piscataway, N.J., 1985, pp. 677–692.

[4] A. Brüggemann. "Regular Expressions into Finite Automata." In: *Theor. Comput. Sci.* 120 (Jan. 1993), pp. 197–213.

[5] B. Cakir and S. Malik. "Reverse Engineering Digital ICs Through Geometric Embedding of Circuit Graphs". In: *ACM Trans. Des. Autom. Electron. Syst.* 23.4 (July 2018), 50:1–50:19. ISSN: 1084-4309. DOI: 10.1145/3193121.

[6] L. Cordella, P. Foggia, C. Sansone, and M. Vento. "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 26 (Nov. 2004), pp. 1367–1372. DOI: 10.1109/TPAMI.2004.75.

[7] R. Duffin. "Topology of series-parallel networks". In: *Journal of Mathematical Analysis and Applications* 10.2 (1965), pp. 303–318. ISSN: 0022-247X. DOI: https://doi.org/10.1016/0022-247X(65)90125-3.

[8] B. Dutertre. *Yices Manual*. July 28, 2016.

[9] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. "Adding regular expressions to graph reachability and pattern queries". In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pp. 39–50.

[10] *Flex: The Fast Lexical Analyzer*. https://github.com/westes/flex.

[11] J. Friedl. *Mastering Regular Expressions*. Aug. 2006. ISBN: 0596528124.

[12] B. Gallagher. "Matching structure and semantics: A survey on graph-based pattern matching". In: *AAAI Fall Symposium - Technical Report* 6 (Jan. 2006).

[13] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik. "Template-based Circuit Understanding". In: *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*. FMCAD '14. Lausanne, Switzerland: FMCAD Inc, 2014, 17:83–17:90. ISBN: 978-0-9835678-4-4.

[14] M. C. Hansen, H. Yalcin, and J. P. Hayes. "Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering". In: *IEEE Design Test of Computers* 16.3 (July 1999), pp. 72–80. ISSN: 0740-7475. DOI: 10.1109/54.785838.

[15] "IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))". In: *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)* (Dec. 2008), pp. 183–195.

[16] S. Kleene. *Representation of Events in Nerve Nets and Finite Automata*. Memorandum (Rand Corporation). Rand Corporation, 1951.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[17] C. Krieg. “Pattern-Based Hardware Trojan Characterization for Design Security Assessment”. PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Jan. 2019.

[18] V. Kumar. “Algorithms for Constraint Satisfaction Problems: A Survey”. In: *A.I. Mag* 13 (Oct. 1998).

[19] S. Kundu, S. Lerner, and R. K. Gupta. “Translation Validation of High-Level Synthesis”. In: *High-Level Verification: Methods and Tools for Verification of System-Level Designs*. New York, NY: Springer New York, 2011, pp. 97–121. ISBN: 978-1-4419-9359-5. DOI: 10.1007/978-1-4419-9359-5_7. URL: https://doi.org/10.1007/978-1-4419-9359-5_7.

[20] J. Larrosa and G. Valiente. “Constraint Satisfaction Algorithms for Graph Pattern Matching.” In: *Mathematical Structures in Computer Science* 12 (Aug. 2002), pp. 403–422. DOI: 10.1017/S0960129501003577.

[21] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. “An in-depth comparison of subgraph isomorphism algorithms in graph databases”. In: *Proceedings of the VLDB Endowment* 6 (Dec. 2012), pp. 133–144. DOI: 10.14778/2535568.2448946.

[22] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia. “WordRev: Finding word-level structures in a sea of bit-level gates”. In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. June 2013, pp. 67–74. DOI: 10.1109/HST.2013.6581568.

[23] W. Li, Z. Wasson, and S. A. Seshia. “Reverse engineering circuits using behavioral pattern mining”. In: *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*. June 2012, pp. 83–88. DOI: 10.1109/HST.2012.6224325.

[24] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. “Strong Simulation: Capturing Topology in Graph Pattern Matching”. In: *ACM Transactions on Database Systems (TODS)* 39 (Jan. 2014), p. 4. DOI: 10.1145/2528937.

[25] C. McCreesh, P. Prosser, C. Solnon, and J. Trimble. “When Subgraph Isomorphism is Really Hard, and Why This Matters for Graph Databases”. In: *J. Artif. Intell. Res.* 61 (2018), pp. 723–759.

[26] T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang. “Gate-level netlist reverse engineering for hardware security: Control logic register identification”. In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. May 2016, pp. 1334–1337. DOI: 10.1109/ISCAS.2016.7527495.

[27] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather. “SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm”. In: *30th ACM/IEEE Design Automation Conference*. June 1993, pp. 31–37. DOI: 10.1145/157485.164556.

[28] M. Olbrich, R. Popp, L. Näthke, L. Hedrich, and E. Barke. “A Combined Structural and Symbolic Method for Automatic Behavioral Modeling of Nonlinear Analog Circuits”. In: (Sept. 2).

[29] *OpenCores*. https://opencores.org/.

[30] *OpenCores project "Logarithm function, base-2, single-cycle"*. https://opencores.org/projects/fast_log. Accessed: 2019-11-26.

[31] *OpenCores project "Simple Asynchronous Serial Controller"*. https://opencores.org/projects/sasc. Accessed: 2019-11-26.

[32] *OpenCores project "USB 2.0 Function Core"*. https://opencores.org/projects/usb. Accessed: 2019-11-26.

[33] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009, pp. 202–239. ISBN: 0136042597.

[34] H. Seba, S. Lagraa, and E. Ronando. “Comparison Issues in Large Graphs: State of the Art and Future Directions”. In: *ArXiv* abs/1502.07576 (2015).

[35] Y. Shi, B. Gwee, Ye Ren, Thet Khaing Phone, and Chan Wai Ting. “Extracting functional modules from flattened gate-level netlist”. In: *2012 International Symposium on Communications and Information Technologies (ISCIT)*. Oct. 2012, pp. 538–543. DOI: 10.1109/ISCIT.2012.6380958.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[36] M. Soeken, B. Sterin, R. Drechsler, and R. Brayton. "Simulation Graphs for Reverse Engineering". In: *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*. FMCAD '15. Austin, Texas: FMCAD Inc, 2015, pp. 152–159. ISBN: 978-0-9835678-5-1.

[37] C. Solnon. "AllDifferent-based filtering for subgraph isomorphism". In: *Artificial Intelligence* 174.12 (2010), pp. 850–864. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2010.05.002.

[38] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascon, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik. "Reverse Engineering Digital Circuits Using Structural and Functional Analyses". In: *IEEE Transactions on Emerging Topics in Computing* 2.1 (Mar. 2014), pp. 63–80. ISSN: 2168-6750. DOI: 10.1109/TETC.2013.2294918.

[39] E. Tashjian and A. Davoodi. "On Using Control Signals for Word-level Identification in a Gate-level Netlist". In: *Proceedings of the 52Nd Annual Design Automation Conference*. DAC '15. San Francisco, California: ACM, 2015, 78:1–78:6. ISBN: 978-1-4503-3520-1. DOI: 10.1145/2744769.2744878.

[40] Z. Terem, G. Kamhi, M. Y. Vardi, and A. Irron. "Pattern search in hierarchical high-level designs". In: *Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems, 2004. ICECS 2004.* IEEE. 2004, pp. 519–522.

[41] *The DOT Language*. https://www.graphviz.org/doc/info/lang.html.

[42] J. R. Ullmann. "An Algorithm for Subgraph Isomorphism". In: *J. ACM* 23.1 (Jan. 1976), pp. 31–42. ISSN: 0004-5411. DOI: 10.1145/321921.321925.

[43] C. Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/.

[44] C. Yu and M. Ciesielski. "Automatic word-level abstraction of datapath". In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. May 2016, pp. 1718–1721. DOI: 10.1109/ISCAS.2016.7538899.

[45] S. Zampelli, Y. Deville, and C. Solnon. "Solving Subgraph Isomorphism Problems with Constraint Programming". In: *Constraints* 15.3 (July 2010), pp. 327–353. ISSN: 1383-7133. DOI: 10.1007/s10601-009-9074-3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Appendices

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Appendix A

State machine Verilog code packet assembly in USB core

Listing A.1: Full listing of the USB core[1] packet assembly state machine, can be found in rtl/verilog/usbf_pa.v

```
always @(state or send_data or tx_ready or tx_valid_r or send_zero_length_r)
   begin
        next_state = state;     // Default don't change current state
        tx_valid_d = 1'b0;
        dsel = 1'b0;
        rd_next = 1'b0;
        last = 1'b0;
        crc_sel1 = 1'b0;
        crc_sel2 = 1'b0;
        case(state)             // synopsys full_case parallel_case
           IDLE:
                   begin
                        if(send_zero_length_r && send_data)
                           begin
                                tx_valid_d = 1'b1;
                                next_state = WAIT;
                                dsel = 1'b1;
                           end
                        else
                        if(send_data)                   // Send DATA packet
                           begin
                                tx_valid_d = 1'b1;
                                next_state = DATA;
                                dsel = 1'b1;
                           end
                   end
           DATA:
                   begin
                        if(tx_ready && tx_valid_r)
                                rd_next = 1'b1;

                        tx_valid_d = 1'b1;
                        if(!send_data && tx_ready && tx_valid_r)
                           begin
```

[1] *OpenCores project "USB 2.0 Function Core"*. https://opencores.org/projects/usb. Accessed: 2019-11-26.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
35                          dsel = 1'b1;
36                          crc_sel1 = 1'b1;
37                          next_state = CRC1;
38                       end
39                end
40          WAIT:               // In case of early tx_ready ...
41                begin
42                    crc_sel1 = 1'b1;
43                    dsel = 1'b1;
44                    tx_valid_d = 1'b1;
45                    next_state = CRC1;
46                end
47          CRC1:
48                begin
49                    dsel = 1'b1;
50                    tx_valid_d = 1'b1;
51                    if(tx_ready)
52                       begin
53                          last = 1'b1;
54                          crc_sel2 = 1'b1;
55                          next_state = CRC2;
56                       end
57                    else
58                       begin
59                          tx_valid_d = 1'b1;
60                          crc_sel1 = 1'b1;
61                       end
62
63                end
64          CRC2:
65                begin
66                    dsel = 1'b1;
67                    crc_sel2 = 1'b1;
68                    if(tx_ready)
69                       begin
70                          next_state = IDLE;
71                       end
72                    else
73                       begin
74                          last = 1'b1;
75                       end
76
77                end
78        endcase
79    end
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct – Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Vienna, Austria April 15, 2020

Martin Mosbeck

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub