Pattern-matching and substitution in electronic circuits

Théophile Bastian, under supervision of Carl-Johan Seger and Mary Sheeran
Chalmers University, Göteborg, Sweden

February – June 2017

Abstract

The present report describes and summarizes my 1st year of Master's degree's internship at the university of Chalmers, Göteborg, Sweden. I worked under supervision of Carl-Johan Seger and Mary Sheeran.

The project's goal was to contribute to VossII, a hardware proving tool cloning fl, developed at Intel by Carl Seger for internal usage only. It led me to work on *pattern-matching* on electronic circuits for a *search-and-replace* method, allowing one to apply previously proved transformations to a circuit.

This problem turns out to be more or less the *subgraph isomorphism problem*, which is NP-complete, and must nevertheless be solved efficiently on processor-sized circuits on this particular case.

During my internship, I developed a C++ library to perform this task that will be integrated in VossII, based on a few well-known algorithms as well as some ad-hoc heuristics and algorithm tweaks to better match the context of circuits. One of my objectives all along was also to keep a clean and well-documented codebase, as the tool will have to be maintainable by others later.

Contents

1 Introduction

In the previous years, verification and proved software has gathered an increasing interest in the computer science community, as people realised how hard bugs are to track down, and the little confidence they had in their own code. But hardware bugs are even more tedious to find and fix, and can easily lead to disastrous consequences, as those cannot be patched on existing hardware. For instance, the well-known Pentium "FDIV" bug [Pra95] that affected a large number of Pentium processors lead to wrong results for some floating point divisions. Intel had to offer to replace all the defective CPUs, leading to an announced loss of 475 million dollars [Nic11]. Even recently, the Skylake and Kaby Lake hyperthreading bug had to be patched using microcode, losing performance and reliability.

To avoid such disasters, the industry nowadays uses a wide range of techniques to catch bugs as early as possible — which, hopefully, is before the product's release date. These techniques include of course a lot of testing on simulated hardware or FPGAs (since an actual processor is extremely expensive to burn). A lot of testing is run as a routine on the current version of the hardware, to catch and notify the designers, since it remains the easiest way to test the behaviour of a circuit. Symbolic trajectory evaluation [HS97] has also its place in the domain, allowing one to run a circuit on a few cycles (before it becomes too expensive) with symbolic values, *ie.* variables instead of zeroes, ones and Xs (for "not a value"). This kind of testing is way more powerful than plain testing, since its results are more precise; yet it is also too expensive to run on a significantly long number of cycles, and therefore a lot of bugs are impossible to catch this way.

The previous methods are great cheap strategies to run the first tests on a circuit, but give only little confidence in its correction — it only proves that among all the cases that were tested, all yielded a correct behaviour. These reasons led to the development of proved hardware in the industry. On circuits as complex as processors, usually, only sub-components are proved correct with respect to a given specification of its behaviour (usually source code that should behave as the processor is expected to behave, itself with respect to the written documentation draft of the circuit). These proofs are typically valid only while the circuit is kept in a specified context, *ie.* a set of valid inputs, tensions, etc. — that should, but is not proved to, be respected by the other parts of the circuit. Yet, this trade-off between proved correctness and engineer's work time already gives a pretty good confidence in the circuit.

In this context, Carl Seger was one of the main developers of fl at Intel [Seg93] [SJO+05] [Seg06], a functional ml-inspired programming language integrating many features useful to get insights of a circuit, testing it and proving it. It mostly features symbolic trajectory evaluation based model checking and theorem proving, and is intended to be an all-purpose toolbox for the hardware prover.

Among other features, it includes a "search and replace" feature, which can search every occurrence of a given gates pattern in a circuit, and replace it by some other gates pattern, proved observationally equivalent beforehand. Time has proved this method very efficient to design circuits: this way, one can start from an inefficient, yet simple circuit, prove it, and then refine it into an equivalent, yet efficient one, through proved transformations. It is also possible to go the other way, and start with an optimized circuit, hard to understand, and make it easier to understand to work more efficiently.

My internship lies amid a project shared between Carl-Johan Seger and Mary Sheeran, aiming to develop tools for proved design of FPGA circuits. One of the keystones of this project is an open-sourced and publicly available version of fl, used for the proving part, and is still at the moment under heavy development.

My part of the work resided on this "search and replace" tool. More specifically, I focused on writing a C++ library, *isomatch*, which is interfaced with the C core of fl and provides it with low-level and very fast functions for this task.

Integer constant $n, m, p, q, \ldots$

Wire $in0, out0, ctl0, \ldots$

Vector $\vec{v}^{\,n}$ *(n elements of type v)*

Circuit $c, d, \ldots ::=$	$\text{delay}(in0, out0)$	*(delay 1 clock tick)*
	$\mid \text{tristate}(in0, out0, ctl0)$	*(three-state gate)*
	$\mid \text{comb}(\vec{in0}^{\,n}, \vec{out0}^{\,m}, \vec{e}^{\,m})$	*(combinatorial gate)*
	$\mid \text{assert}(\vec{in0}^{\,n}, \vec{e}^{\,m})$	*(assertion gate)*
	$\mid \text{group}(\vec{in0}^{\,n}, \vec{out0}^{\,m}, \vec{c}^{\,p})$	*(circuit hierarchical group)*
Binary operator $\otimes ::=$	$\wedge$	*(and)*
	$\mid \vee$	*(or)*
	$\mid \oplus$	*(xor)*
	$\mid +$	*(add)*
	$\mid -$	*(sub)*
	$\mid \times$	*(times)*
	$\mid \div$	*(div)*
	$\mid \%$	*(mod)*
	$\mid \ll$	*(logical shift left)*
	$\mid \gg_{\text{l}}$	*(logical shift right)*
	$\mid \gg_{\text{a}}$	*(arithmetic shift right)*
Unary and constant operator $\otimes_0 ::=$	$\ll_0$	*(logical shift left of constant)*
	$\mid \gg_{0,\text{l}}$	*(logical shift right of constant)*
	$\mid \gg_{0,\text{a}}$	*(arithmetic shift right of constant)*
Unary operator $\ominus ::=$	$\neg$	*(logical negation)*
Expression $e, f, \ldots ::=$	x	*(variable)*
	$\mid n$	*(integer constant)*
	$\mid e \otimes f$	*(binary operator)*
	$\mid e \otimes_0 n$	*(unary operator with constant)*
	$\mid \ominus e$	*(unary operator)*
	$\mid e_{\mid\, n \ldots m}$	*(slicing: take a subword)*
	$\mid e \mid f$	*(merging: concatenate two words)*

Figure 1: AST of circuits used

2 Problem setting

2.1 Circuit description

The circuits on which *isomatch* is working are described, and internally represented, by the AST in Figure 1.

The most important thing in the description of circuits here, is that those circuits are organized as a hierarchy of *circuit groups*. This hierarchy can be seen as the construction of a circuit by assembling smaller integrated circuits (ICs), themselves built the same way, etc. A group is composed of sub-circuits, input pins and output pins. Each level can of course contain "leaf" gates, like *and* or *delay* gates. This is important, because it allows the program to work on smaller areas of the circuit (*eg.* loading in memory only a part of the circuit, etc.).

Isomatch comes along with a small parser for a toy ad-hoc language, designed to allow one to quickly write a test circuit and run the *isomatch* functions on it. There was no real apparent need for a better language, or a standard circuit description language (such as VHDL [Nav97]), since the user will mostly use *isomatch* through fl, and feed it directly with data read from fl — which is able to handle *eg.* VHDL.

2.2 Codebases

Carl Seger's new version of fl is currently being developed as **VossII**, and is yet to be open-sourced. My contribution to the project, **isomatch**, is free software, and is available on GitHub:

https://github.com/tobast/circuit-isomatch/

2.3 Problems

More precisely, the problems that *isomatch* must solve are the following.

1. Given two circuit groups, are they structurally equivalent? That is, are they the same circuit, arranged in a different way, with possibly different names, etc.?

2. Given two circuits, *needle* and *haystack*, find every (non-overlapping) occurrence of *needle* in *haystack*. An occurrence is a set S of sub-circuits of *haystack* such that there is a one-to-one mapping of structurally equivalent circuits of S with circuits of *needle*, and those circuits are connected the same way in both circuits.

Both problems are hard. The first one is an instance of the graph isomorphism problem, as the actual question is whether there exists a one-to-one mapping between sub-circuits of the two groups, such that every mapped circuit is equal to the other (either directly if it is a leaf gate, or recursively with the same procedure); and whether this mapping respects connections (edges) between those circuits. Graph isomorphism is known to be in NP (given a permutation of the first graph, it is polynomial to check whether the first is equal to the second *wrt.* the permutation), but not known to be in either P or NP-complete. Thus, since Babai's work on graph isomorphism [Bab16] is only of theoretical interest (at the moment), the known algorithms remain in worst-case exponential time, and require ad-hoc heuristics for specific kind of graphs to get maximum efficiency.

The second one is an instance of subgraph isomorphism problem, which is known to be NP-complete [Coo71]. Even though a few algorithms (discussed later) are known to be efficient in most cases for this problem, it is nevertheless necessary to implement them the right way, and with the right heuristics, to get the desired efficiency for the given problem.

2.4 Code quality

Another prominent objective was to keep the codebase as clean as possible. Indeed, this code will probably have to be maintained for quite some time, and most probably by other people than me. This means that the code and all its surroundings must be really clean, readable and reusable. I tried

to put a lot of effort in making the code idiomatic and easy to use, through *eg.* the implementation of iterators over my data structures when needed, idiomatic C++14, etc.

This also means that the code has to be well-documented: the git history had to be kept clean and understandable, and a clean documentation can be generated from the code, using `doxygen`. The latest documentation is also compiled as HTML pages here:

`https://tobast.fr/m1/isomatch`

Since the code is C++, it is also very prone to diverse bugs. While I did not took the time to integrate unit tests — which would have been a great addition —, I used a sequence of test that can be run using `make test`, and tests a lot of features of isomatch.

The code is also tested regularly and on a wide variety of cases with `valgrind` to ensure that there are no memory errors — use-after-free, unallocated memory, memory leaks, bad pointer arithmetics, ... In every tested case, strictly no memory is lost, and no invalid read was reported.

2.5 Sought efficiency

The goal of *isomatch* is to be applied to large circuits on-the-fly, during their conception. Those circuits can (and will probably) be as large as a full processor, and the software will be operated by a human, working on their circuit. Thus, *isomatch* must be as fast as possible, since matching operations will be executed quite often, and often multiple times in a row. It must then remain fast enough for the human not to lose too much time, and eventually lose patience.

3 General approach

3.1 Theoretical solution

The global strategy used to solve efficiently the problem can be broken down to three main parts.

Signatures. The initial idea to make the computation fast is to aggregate the inner data of a gate — be it a leaf gate or a group — in a kind of hash, a 64 bits unsigned integer. This approach is directly inspired from what was done in fl, back at Intel. This hash must be easy to compute, and must be based only on the structure of the graph — that is, must be entirely oblivious of the labels given, the order in which the circuit is described, the order in which different circuits are plugged on a wire, ... The signature equality, moreover, must be sound; that is, two signatures must necessarily be equal if the circuits are indeed equal.

This makes it possible to rule out quickly whether two circuits are candidates for a match or not, and run the costy actual equality algorithm on fewer gates.

Group equality. The group equality algorithm is a standard backtracking algorithm. It tries to build a match between the graphs by trying the diverse permutations of elements with the same signature. It can also communicate with the signing part, to request a more precise (but slightly slower to compute) signature when it has too many permutations to try.

This part could be enhanced, but does not slow down the algorithm on the tested examples, so I focused on other parts.

Pattern matching. This part is the one responsible to answer queries for occurrences of a sub-circuit in a circuit. It uses extensively the signatures to determine whether two circuits could be a match or not before spending too much time actually finding matches, but cannot rely on it as heavily as group equality, since only the first level of precision is applicable here (detailed later).

This part mostly consists in an implementation of Ullmann's algorithm for subgraph isomorphism [Ull76], a well-known algorithm for this problem, that uses the specificities of the graph to be a little faster.

3.2 Workflow

In a first time, to get the algorithms, heuristics and methods right, I designed a prototype in OCaml. This prototype was not intended to — and neither was — optimized, but allowed me to find places where the program took an unacceptable amount of time. For instance, I left the prototype that I thought fast enough to compute a group equality a whole night long, before finding out in the morning it was actually not fast enough at all. This first version was also written in a strongly typed language, with a lot of static guarantees. It gave me enough confidence in my methods and strategies to move on to a way faster and optimized C++ version, the current version of *isomatch*.

4 Signatures

The signature is computed as a simple hash of the element, and is defined for every type of expression and circuit. It could probably be enhanced with a bit more work to cover more uniformly the hash space, but no illegitimate collision (that is, a collision that could be avoided with a better hash function, as opposed to collisions due to an equal local graph structure) was observed on the examples tested.

Signature constants. Signature constants are used all around the signing process, and is a 5-tuple $\mathcal{SC} = (a, x_l, x_h, d_l, d_h)$ of 32 bits unsigned numbers. All of x_l, x_h, d_l and d_h are picked as prime numbers between 10^8 and 10^9 (which just fits in a 32 bits unsigned integer); while a is a random integer uniformly picked between 2^{16} and 2^{32}. These constants are generated by a small python script, `util/primegen/pickPrimes.py` in the repository.

Those constants are used to produce a 64 bits unsigned value out of another 64 bits unsigned value, called v thereafter, through an operator $\mathfrak{h}$, computed as follows (with all computations done on 64 bits unsigned integers).

function $\mathfrak{h}(\mathcal{SC}, v)$
  $out1 \leftarrow (v + a) \cdot x_l$
  $v_h \leftarrow (v \gg_\mathrm{l} 32) \oplus (out1 \gg_\mathrm{l} 32)$
  $low \leftarrow out1 \,\%\, d_l$
  $high \leftarrow ((v_h + a) \cdot x_h) \,\%\, d_h$
  return $low + 2^{32} \cdot high$
end function

Expressions. Each type of expression (or, in the case of expression with operator, each type of operator) has its signature constant, $\mathcal{SC}_{\mathrm{exprType}}$. The signature of a commutative expression in its operands is always commutative, and the signature of a non-commutative expression should not be (and is not, except for collisions). The value v used to sign the expression (in $\mathfrak{h}(\mathcal{SC}_{\mathrm{exprType}}, v)$) is then the sum (respectively difference) of the signature of its parameters for commutative (respectively non-commutative) expressions.

Circuits' inner signature. Every circuit is associated with a value describing its *type* (rather than its contents): 8 bits of circuit type ID (delay, tristate, ...), the number of inputs on the next 8 bits, and the number of outputs on 8 more bits. This value is then xored with the inner value of the circuit: for a combinatorial gate, the xor of its expressions' signatures; for a group, the sum of its children's signatures[1], ... This value constitutes the circuit's *inner signature*.

Circuits' signature of order n. The inner signature does not capture at all the *structure* of the graph. An information we can capture without breaking the signature's independence towards the order of description of the graph, is the set of its neighbours. Yet, we cannot "label" the gates without breaking this rule; thus, we represent the set of neighbours by the set of the *neighbours' signatures*.

[1] As a group is likely to have multiple occurrences of a single identical circuit, it would be unwise to xor its children's signatures, even though the usual advice is to combine hashes by xoring them.

At this point, we can define the *signature of order n* ($n \in \mathbb{N}$) of a circuit C as follows:

$$\begin{aligned}\mathrm{sig}_0(C) &:= \text{inner signature of } C \\ \mathrm{sig}_{n+1}(C) &:= \text{inner signature of } C + \text{IO adjacency} + \sum_{C_i \in \text{ neighbours of inputs}} \mathrm{sig}_n(C_i) - \sum_{C_o \in \text{ neighbours of outputs}} \mathrm{sig}_n(C_o)\end{aligned}$$

The "IO adjacency" term is an additional term in the signatures of order above 0, indicating what input and output pins of the circuit group containing the current gate are adjacent to it. Adding this information to the signature was necessary, since a lot of gates can be signed differently using this information (see Corner cases in Section 7.2).

The default order of signature used in all computations, unless more is useful, is 2, after a few benchmarks.

Efficiency. Every circuit memoizes all it can concerning its signature: the inner signature, the IO adjacency, the signatures of order n already computed, etc.

This memoization, alongside with the exclusive use of elementary operations, makes the computation of a signature very fast. The computation is linear in the number of gates in a circuit, times the order computed; the computation is lazy.

To keep those memoized values up to date whenever the structure of the circuit is changed (since this is meant to be integrated in a programming language, fl, a standard workflow will possibly be create a circuit, check its signature, alter it, then check again), each circuit keeps track of a "timestamp" of last modification, which is incremented whenever the circuit or its children are modified. A memoized data is always stored alongside with a timestamp of computation, which invalidates a previous result when needed.

One possible path of investigation for future work, if the computation turns out to be still too slow in real-world cases — which looks unlikely, unless fl's substitution is run on a regular basis for a huge number of cases using *eg.* a crontab for automated testing —, would be to try to multithread this computation.

5 Group equality

Given two circuit group gates, the task of group equality is to determine whether the two groups are structurally equivalent, as discussed above.

Group equality itself is handled as a simple backtracking algorithm, trying to establish a match (an isomorphism, that is, a permutation of the gates of one of the groups) between the two groups given.

The gates of the two groups are matched by equal number of inputs and outputs and equal signatures — based on the signature of default order (that is, 2). A few checks are made, *eg.* every matching group must have the same size on both sides (if not, then, necessarily, the two groups won't match). Then, the worst case of number of permutations to check is evaluated.

If this number is too high, the signature order will be incremented, and the matching groups re-created accordingly, until a satisfyingly low number of permutations is reached (or the diameter of the circuit is reached, meaning that increasing the order of signature won't have any additional impact). This order increase "on-demand" proved itself very efficient, effectively lowering the number of permutations examined to no more than 4 in studied cases.

Once a permutation is judged worth to be examined, the group equality is run recursively on all its matched gates. If this step succeeds, the graph structure is then checked. If both steps succeed, the permutation is correct and an isomorphism has been found; if not, we move on to the next permutation.

6 Pattern-match

We finally need to be able to find every occurrence of a given *needle* circuit in a bigger *haystack* circuit — at any level in its groups hierarchy. This problem is basically graph isomorphism with some

specificities, and for this purpose I used a classical algorithm for the subgraph isomorphism problem, *Ullmann*.

6.1 Ullmann's algorithm

One of the classical algorithms to deal with the subgraph isomorphism problem was first described by Julian R Ullmann in 1976 [Ull76]. Another, more recent algorithm to deal with this problem is Luigi P Cordella's VF2 algorithm [CFS+04], published in 2004. This algorithm is mostly Ullmann's algorithm, transcribed in a recursive writing, with the addition of five heuristics. I originally planned to implement both algorithms and benchmark both, but had no time to do so in the end; though, Ullmann with the few additional heuristics applicable in our very specific case turned out to be fast enough.

Ullmann is a widely used and fast algorithm for this problem. It makes an extensive use of adjacency matrix description of the graph, and the initial article takes advantage of the representation of those matrices as bitsets to make extensive use of bitwise operations.

The to-be-built permutation matrix is a $|needle| \times |haystack|$ matrix. Each 1 in a cell (i, j) indicates that the i-th needle part is a possible match with the j-th haystack part. This matrix is called *perm* thereafter.

The algorithm, left apart the REFINE function, which is detailed just after and can be omitted for a (way) slower version of the algorithm, is described in Figure 2.

```
function FIND_AT_DEPTH(depth, perm, freeVert)
    if no 1s on perm[depth] then
        return
    end if
    Save perm
    for 0 ≤ chosen < |haystack| such that perm[depth][chosen] = 1 and freeVert[chosen] do
        Put 0s everywhere on perm[depth], but on chosen
        Refine perm
        if a row of perm has only 0s then
            return
        end if
        if depth = |needle| − 1 then
            Store perm as a result
        else
            FIND_AT_DEPTH(depth+1, perm, freeVert with freeVert[chosen] = 0)
        end if
        Restore perm
    end for
end function

function FIND(perm)
    return FIND_AT_DEPTH(0, perm, [1, . . . , 1])
end function
```

Figure 2: Ullmann's algorithm (without refining)

The refining process is the actual keystone of the algorithm. It is the mechanism allowing the algorithm to cut down many exploration branches, by changing ones to zeroes in the matrix being built.

The idea is that a match between a needle's vertex i and a haystack's vertex j is only possible if, for each neighbour k of i, j has a neighbour k' such that the permutation matrix has a one in position (k, k'). In other words, a match between i and j is only possible if every neighbour k of i (in needle) has a possibly matching (*wrt. perm*) vertex k' (in haystack) which is a neighbour of j.

For instance, for the needle from Figure 3's, we can try to check the ones corresponding to the vertex a (the coloured one). Refining it while matching it with Figure 4 will leave a 1 on the match $a - a'$, since every neighbour of a can be matched with a neighbour of a' (which, hopefully, are a 1 in the matrix): b matches y and c matches z, for instance. It is not the case with the haystack from Figure 5: if the process went far enough already, there should be no corresponding vertex for either b or c, since there are no such two vertices with an edge linking them. If there is indeed no match at this point for either w or c, the 1 in the cell matching $a - a'$ will be turned to a 0.

This condition is checked on every 1 in the permutation matrix. If it is not met, the cell is nulled. This, though, potentially creates new ones not matching the condition: the process must be run again, until no new zeroes appear.

In the initial article [Ull76], Ullmann advocates for bitwise tricks to complete this expensive step: indeed, checking the existence of such a k' can be done by checking the nullity of the bitwise AND of the adjacency of j and the permutation matrix row of k.

The refining function is detailed in Figure 7.

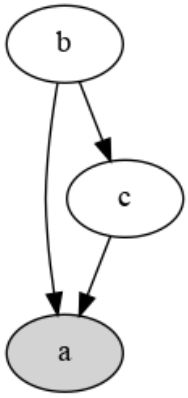

Figure 3: Needle

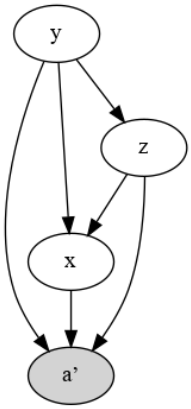

Figure 4: Matching haystack

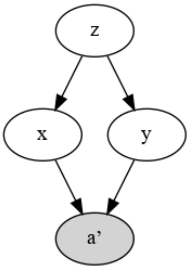

Figure 5: Non-matching haystack

Figure 6: Example: Ullmann refining process

```
function REFINE(perm)
    while changes during last run do
        for each needle vertex i do
            for each haystack vertex j do
                if perm[i][j] = 1 then
                    for each neighbour k of i in needle do
                        if perm[k] & haystack_adjacency[j] = 0 then
                            perm[i][j] ← 0
                        end if
                    end for
                end if
            end for
        end for
    end while
end function
```

Figure 7: Ullmann's refining function

6.2 Ullmann for *isomatch*

Graph used. Our circuit is not actually a graph just as-is: indeed, a wire can be connected to multiple circuits (multiple gates' inputs, or even multiple gates' outputs when using tristate circuits).

This could be transposed into a graph with $\frac{n(n-1)}{2}$ edges (the complete subgraph) for this particular wire. Though, internally, a wire is better represented as a vertex itself, with n edges linking it to the connected gates. This representation is also used in Ullmann's implementation, leading to a permutation matrix of (|needle gates| + |needle wires|) × (|haystack gates| + |haystack wires|).

Final result. Once a result (*ie.* a correct permutation) is obtained, we further need to check it is actually a solution of our problem. Indeed, while the structure is guaranteed by the algorithm to be the same, we still need to check that every circuit is equal to its matched one, through the procedure described in Section 5. So far, only the equality of signatures was checked. We only need to check the circuits, as the wires are necessarily actually matching.

Non-overlapping results. We want our results to be non-overlapping (because we won't be able to perform a search-and-replace if it is not the case). Whenever two potential results are conflicting, an arbitrary one of the two can be returned (a human user is operating the software and can make a narrower search if needed).

To match this specification, we must keep track of the circuits that are already included in a match. We also cannot include an ancestor of a circuit that was included in a match in another match (though this is not possible, because the needle can't be included in itself, but signature collisions could occur).

6.3 Implementation optimisations

Pre-check. The needle will, in most cases, not be found at all in a given hierarchy group of the haystack. To avoid wasting computation time, we first check that every signature present in the needle is present at least as many times in the haystack. This simple check saved a lot of time.

Initial permutation matrix. The matrix is first filled according to the signatures' matches. Note that only signatures of order 0 — *ie.* the inner data of a vertex — can be used here: indeed, we cannot rely on the context here, since there can be some context in the haystack that is absent from the needle, and we cannot check for "context inclusion" with our definition of signatures: *all* the context must be exactly the same for two signatures to match. It is then refined a bit more, by making sure that for every match, every potentially matching gate has the same "wire kinds". Indeed, a gate needle's wire must have at least the same inbound adjacent signatures as its matching haystack wire, and same goes for outbound adjacent signatures. Thus, two circuits cannot be matched if this condition is not respected for each pair of corresponding wires of those circuits, and their corresponding cell in the permutation matrix can be nulled.

Conversion to adjacency matrix. The internal structures and graphs are represented as inherited classes of `CircuitTree`, connected to various `WireId`s. Thus, there is no adjacency matrix easily available, and the various vertices have no predefined IDs. Thus, we need to assign IDs to every vertex, *ie.* every gate and wire.

Order of rows and columns In his original paper, Ullmann recommends to index the vertices in order of decreasing degree (*ie.*, the vertices with highest degree are topmost/leftmost in the matrix). This amplifies the effect of the refinement procedure, since vertices with higher degree are connected to more vertices, and thus has a higher chance of having a neighbour with no matching neighbour in the haystack. This allows the algorithm to cut whole branches of the search tree very early, and speeds up the algorithm a lot. Yet, Ullmann also states that on specific cases, a different order might be more appropriate.

My idea was that it might be interesting to put first all the wires, and then all the circuits, or the other way around. For that, I did a few benchmarks. The measures were made on a system with a nearly-constant load during the experiments. The machine had a i7-6700 processor (3.6GHz reported frequency). All the measures are averaged over 100 runs, and were measured on the usual test set.

Ordering method	Run time (ms)	Loss (%)
Wires by degree decreasing, then gates as they come	48.8	—
As they come, gates then wires	49.1	0.6%
By degree decreasing, wires then gates	49.3	1.0%
As they come, wires then gates	49.3	1.0%
Gates as they come, then wires by degree decreasing	49.5	1.4%
By degree decreasing, all mixed	49.5	1.4%

The time loss is nearly insignificant, and can be explained by constant costs: when we want to sort vertices, the vector must be copied then sorted, which can be more expensive than just taking its elements as they come, if the gain is not high enough.

Nevertheless, I chose to implement the fastest option with respect to this benchmark. If the gap is nearly insignificant, the choice can't really be drastically harmful in the end.

7 Performance

In this section, all the measures were made on a computer with an Intel i7-3770 CPU (3.40GHz) and 8 GB of RAM.

7.1 Small processor

The example I used widely during my project to test my program and check its efficiency periodically was a small processor designed one year earlier as a school project [BCC16]. The processor implements a large subset of ARM. It does not feature a multiplier, nor a divider circuit, but supports instructions with conditions (ARM-flavoured: a whole lot of conditional prefixes can be plugged into an assembly instruction). It was conceived as a few recursively hierarchized modules, but was flattened as a plain netlist when generated from its python code, so I first had to patch its generator code to make its hierarchy apparent.

The circuit is composed, at its root level, of a few modules: a memory unit (access to RAM), a flags unit (handling the ALU's flags), two operands units (decoding the operands and applying a barrel shifter if needed), an opcode decoding unit (decoding the program's opcodes), a registers unit (containing the registers) and a result selector (selecting the output between the ALU's and the registers' output, based on the opcode).

The processor, in the end, has around 2000 leaf gates (but works at word level) and 240 hierarchy groups.

Signature. First, the time required to sign the whole circuit with different levels of signature (*ie.* the order of signature computed for every part of the circuit). In practice, we never compute high order signatures for a whole circuit, as signature of subgroups are always computed by default at the order 2, unless this particular group needs a more accurate signature.

The measures were made for 100 consecutive runs of the program (then averaged for a single run) and measured by the command `time`. The computing time necessary for different signature levels is plotted in Figure 8.

The computation time is more or less linear in in the level of signature required, which is coherent with the implementation. In practice, only small portions of a circuit will be signed with a high order, which means that we can afford really high order signatures (*eg.* 40 or 50, which already means that the diameter of the group is 40 or 50) without having a real impact on the computation time.

This linearity means that we can increase the signature order without too much impact, as we do when computing a group equality.

Equality. To test the circuit group equality, a small piece of code takes a circuit, scrambles it as much as possible — without altering its structure —, *eg.* by renaming randomly its parts, by randomly changing the order of the circuits and groups, ... The circuit is then matched with its unaltered counterpart.

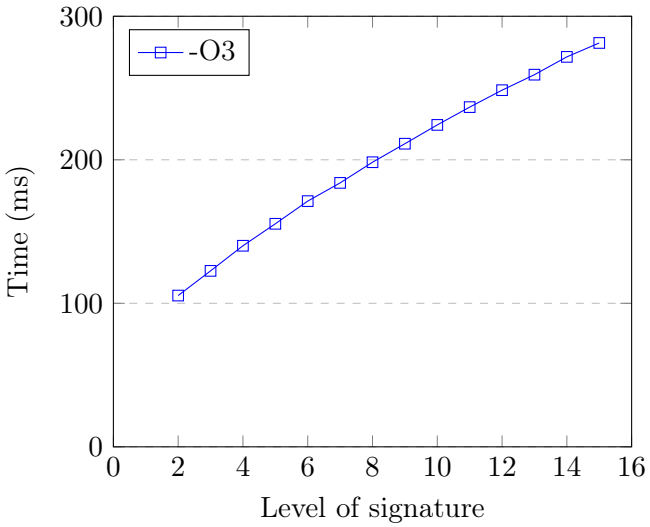

Figure 8: Signature time of the processor for different levels of signature

For the processor described above, it takes about **313 ms** to prove it equal to its scrambled version, and then the other way around. Yet, the memoized results (essentially the signatures) are kept for the second one, considerably speeding it up: the same program proving only one way takes about **310 ms**.

Some signatures of order higher than two need to be computed, and there is a constant time needed to scramble the circuit, etc., which means that the actual equality match time is ridiculously small compared to the signature computation time.

Match. The subcircuit match feature was tested by trying to find every occurrence of a pattern that can be easily found using tools like `grep`. For this purpose, the ad-hoc implementation of a MUX gate was used: two tristate gates and a NOT gate, as in Figure 9.

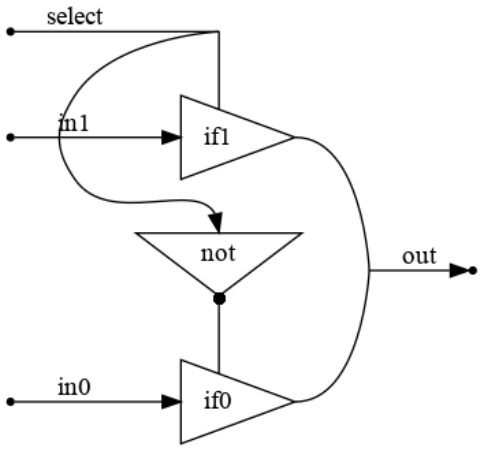

Figure 9: MUX gate made out of tristate and not gates

This group appears 73 times in the processor. To match them all (none are overlapping), it takes **113 ms**.

7.2 Corner cases

There were a few observed cases where the algorithm tends to be slower on certain configurations, and a few other such cases that could be fixed.

I/O pins. In Section 4, we introduce a term named *IO adjacency* in the signatures of order higher than 0. This is because some sub-circuits can be told apart from their signatures only through this

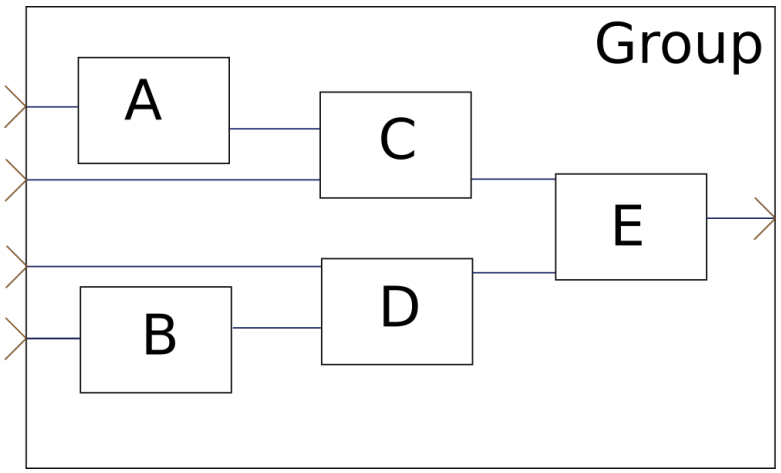

Figure 10: A case where the I/O adjacency term is necessary

information, and the absence of this term slowed down considerably the program before its introduction. This is the case for instance in circuits like the one in Figure 10.

Here, if the adjacent input and output pins of Group are not taken into account in the signatures of the gates (A, B, C, D, E), there is apparently no way to differentiate A from B and C from D. The example can of course be pushed further to way more dramatic cases, for instance with a (binary) tree of circuits in the same shape as above.

Split/merge trees. A common pattern that tends to slow down the algorithm is split/merge trees. Those patterns occur when one wants to merge n one bit wires into a single n bits wire, or the other way around.

These patterns are pretty common, for instance when an opcode is run through a MUX tree to perform the requested operation.

Though, this pattern generates a lot of collisions in signatures. Indeed, for a tree of depth *eg.* 8, a node just below the root will need a signature of order 7 to have a different signature than another one at the same depth. With a signature of order up to 6, only other gates from the tree will be included in the signature when going down in the tree; the exact same gates will be included above the tree's root. Thus, nothing will differentiate one gate from another while the boundary of the tree is not reached (assuming the gates below the tree's leaves are not all the same; if so, more levels will be needed).

As the notion of "left child" and "right child" cannot be used (since it would rely on the order or description of the graph), there seems to be no good way to discriminate those two nodes. Furthermore, the nodes are not totally interchangeable: indeed, when checking for an equality between two such trees, it does not matter which node is the left one; but once this is fixed, the nodes on the layer below cannot be freely exchanged.

For instance, in Figure 11, the orange borders are the boundaries of what can be taken into account for the signatures of order 1 of the gates marked with a red dot. Thus, those signatures are exactly the same.

Conclusion

At this point, *isomatch* seems to be fast enough to be plugged into VossII, and is being integrated at the moment. On all the cases tested — with tests that tried to be as complete as possible by testing each independent feature — it returned a correct result. Even though there are a handful of ways to enhance it, make it faster, etc., it is useable in its current state.

This internship led me to develop new strategies to bypass corner cases where the heuristics were inefficient, the methods inadequate, ... But this project also made me practice again with C++, which I had left behind for some time; and forced me to try to have a code as clean as possible, challenging

me on small details that were easy to implement, but hard to implement in an understandable and bug-proof way.

References

[Bab16] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 684–697. ACM, 2016.

[BCC16] Théophile Bastian, Noémie Cartier, and Nathanaël Courant. "système digital" project. https://github.com/tobast/sysdig-full/, 2016.

[CFS+04] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[HS97] Scott Hazelhurst and Carl-Johan H Seger. Symbolic trajectory evaluation. In *Formal hardware verification*, pages 3–78. Springer, 1997.

[Nav97] Zainalabedin Navabi. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[Nic11] Thomas Nicely. Pentium fdiv flaw faq. 2011. URL: http://www.trnicely.net/pentbug/pentbug.html (visited on 08/10/2017).

[Pra95] Vaughan Pratt. Anatomy of the pentium bug. *TAPSOFT'95: Theory and Practice of Software Development*:97–107, 1995.

[Seg06] Carl Seger. The design of a floating point execution unit using the integrated design and verification (idv) system (abstract only). In *Int. Workshop on Designing Correct Circuits*, 2006.

[Seg93] Carl-Johan H Seger. Voss: a formal hardware verification system user's guide, 1993.

[SJO+05] C-JH Seger, Robert B Jones, John W O'Leary, Tom Melham, Mark D Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, 2005.

[Ull76] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

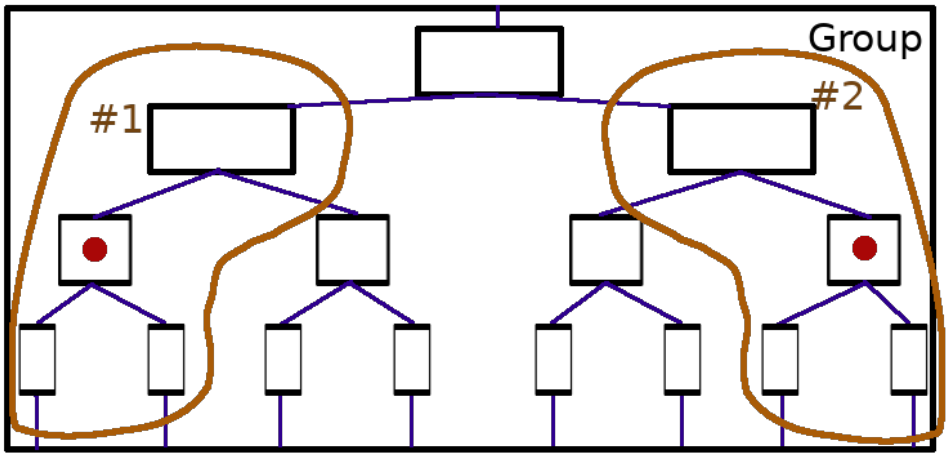

Figure 11: Case of a split (or merge) tree