BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EMPLOYING APPROXIMATE EQUIVALENCE FOR DESIGN OF APPROXIMATE CIRCUITS

VYUŽITÍ PŘIBLIŽNÉ EKVIVALENCE PŘI NÁVRHU PŘIBLIŽNÝCH OBVODŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. JIŘÍ MATYÁŠ**
AUTOR PRÁCE

SUPERVISOR — **RNDr. MILAN ČEŠKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Matyáš Jiří, Bc.**
Obor: Inteligentní systémy
Téma: **Využití přibližné ekvivalence při návrhu přibližných obvodů**
Employing Approximate Equivalence for Design of Approximate Circuits
Kategorie: Formální verifikace

Pokyny:

1. Seznamte se s myšlenkou využití evolučních technik pro automatickou syntézu přibližných aritmetických a logických obvodů a s existujícími konceptuálními návrhy využití technik formální verifikace pro posuzování míry shody přesného a přibližného obvodu.
2. Seznamte se s různými existujícími nástroji pro řešení SAT a SMT problémů, které jsou potenciálně využitelné v daném kontextu.
3. Zkonkretizujte do experimentálně ověřitelné podoby vybrané přístupy k využití formálních technik v daném kontextu a implementujte je.
4. Experimentálně ověřte na vhodně zvolených obvodech různé uvažované přístupy a podrobně diskutujte jejich výhody a nevýhody.

Literatura:

- Holík, L., Lengál, O., Rogalewicz, A., Sekanina, L., Vašíček, Z., Vojnar, T.: Towards Formal Relaxed Equivalence Checking in Approximate Computing Methodology, In: Proc. of WAPCO'16, HiPEAC, 2016.
- Vizel, Y., Weissenbacher, G., Malik, S.: Boolean Satisfiability Solvers and Their Applications in Model Checking, Proceedings of the IEEE, 103(11), 2015.
- Chandrasekharan, A., Soeken, M., Grosse, D., Drechsler, R.: Precise error determination of approximated components in sequential circuits with model checking, In: Proc. of DAC '16, ACM, 2016.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání a alespoň začátek práce na bodě třetím.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Češka Milan, RNDr., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract

This thesis is concerned with the utilization of formal verification techniques in the design of the functional approximations of combinational circuits. We thoroughly study the existing formal approaches for the approximate equivalence checking and their utilization in the approximate circuit development. We present a new method that integrates the formal techniques into the Cartesian Genetic Programming. The key idea of our approach is to employ a new search strategy that drives the evolution towards promptly verifiable candidate solutions. The proposed method was implemented within ABC synthesis tool. Various parameters of the search strategy were examined and the algorithm's performance was evaluated on the functional approximations of multipliers and adders with operand widths up to 32 and 128 bits respectively. Achieved results show an unprecedented scalability of our approach.

Abstrakt

Tato práce je zaměřena na využití formálně verifikačních technik pro návrh funkčních aproximací kombinačních obvodů. Jsou zde důkladně prostudovány existující formální přístupy pro zkoumání přibližné ekvivalence a jejich použití při vývoji aproximovaných obvodů. V rámci této práce je navržena nová metoda, která integruje vybrané formální techniky do Kartézského genetického programování. Klíčovým bodem nového přístupu je využití prohledávací strategie, která vede evoluci směrem k řešením, která lze rychleji verifikovat. Navržený algoritmus byl implementován v rámci syntézního nástroje ABC. Jeho výkonnost byla otestována na vývoji funkčních aproximací násobiček a sčítaček s šířkami vstupních operandů 32, respektive 128 bitů. Dosažené výsledky ukazují výjimečnou škálovatelnost navržené metody.

Keywords

Approximate circuits, relaxed equivalence, evolutionary circuit design, Cartesian genetic programming, ABC.

Klíčová slova

Aproximované obvody, přibližná ekvivalence, evoluční návrh obvodů, Kartézské genetické programování, ABC.

Reference

MATYÁŠ, Jiří. *Employing Approximate Equivalence for Design of Approximate Circuits.* Brno, 2017. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Milan Češka, Ph.D.

Employing Approximate Equivalence for Design of Approximate Circuits

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Dr. Milan Češka. The crucial information about evolutionary algorithms, experimental settings and circuit synthesis was provided by Dr. Zdeněk Vašíček and Ing. Vojtěch Mrázek. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Jiří Matyáš
May 24, 2017

Acknowledgements

I would like to thank Dr. Milan Češka for supervising and his active and professional approach to this thesis, Dr. Zdeněk Vašíček for providing information about evolutionary algorithms and Ing. Vojtěch Mrázek for the enormous help with the experimental evaluation of the results of this thesis.

Contents

Chapter 1

Introduction

Approximate computing has been recently established as a new research field in computer science. Its main purpose is to examine, how computer systems could be made better by relaxing the requirement, that they are always performing exactly correct computations. Approximate computing can be employed in so called *error resilient* applications. These applications can produce acceptable results despite the fact that their underlying computations are performed imperfectly (with errors). This can be caused by various factors: errors average out, errors are not recognizable by limited human perception capabilities, no golden solution is available for validation of the results or users are willing to accept some inaccuracies under special circumstances. The practical applications featuring these attributes include image and multimedia processing, signal processing, data mining, machine learning, neural networks and scientific computing. For example, Google uses this approach in their Tensor processing unit [8]. In general, most of these applications are highly computation intensive and their implementations consume significant amount of energy.

There are three major approaches for performing approximate computing. *Approximate circuits*, such as adders or multipliers, can reduce hardware size or computation time. *Approximate storage* stores approximate values of data (e.g. by truncating less significant bits) instead of saving the exact values or uses less reliable memory. *Software approximation* includes techniques such as loop perforation [14] (some loop iterations viewed as unnecessary can be skipped), memoization, task skipping (skipping tasks similarly to loop skipping) and Monte Carlo algorithms [15]. This thesis further focuses on the first method mentioned – approximate circuits.

There are different techniques for approximate circuit design: over-scaling, over-clocking and functional approximation. This thesis aims at functional approximation where we try to approximate the logic functions computed by the correct circuits. The approximate circuits are designed in such a way that their results are sufficiently similar to the original specification.

Developing accurate and effective approximate circuits is a very challenging task. Basic techniques are usually limited to small combinational and sequential circuits [12, 22] (tens to hundreds of logical gates) and struggle to develop larger circuits, where the evaluation of circuit accuracy becomes computationally unfeasible [25]. There are many different approaches improving the development of approximate circuits that have been examined recently. New methodologies, such as SALSA [27] or SASIMI [26], start with a golden circuit specification and a quality constraint that defines the amount of error to be introduced into the implementation. SALSA then synthesizes an approximate version of the circuit by modifying the original version. The final circuit has to satisfy the defined constraints.

This thesis focuses on approximate circuit design based on evolutionary algorithms [2]. This approach relies on the iterative generation and evaluation of a huge number of candidate solutions. It is crucial to be able to quickly assess the quality (fitness) of a candidate solution in order to find useful approximate circuits. Conventional approaches to fitness evaluation compare the results of the candidate solution to the expected golden results for all possible input combinations. Even though this process can be sped up significantly using a parallel evaluation technique, the task is still unfeasible for larger circuits. For example, with 32-bit multiplier, we need to evaluate 2^{64} different input combinations which is a computationally unfeasible task. To solve this problem, a statistical approach evaluating only a subset of all possible inputs can be utilized. However, this technique called random simulation does not provide any guarantees on the approximation error.

To solve this problem, we employ formal verification methods. Formal verification is used to prove or disprove correctness of a system with respect to a given specification using rigorous mathematical methods. We try to convert the problem of fitness evaluation into a formal description and then employ the verification methods to assess the value of the fitness function. In order to determine the error of a candidate solution, we encode both the candidate and the golden solution into an instance of the formal problem. The two solutions are combined using a special construction – miter [5, 19]. The simplest of miters performs the strict equivalency check, which is not suitable for the approximate circuit development. Therefore more complex miters performing relaxed equivalence checking are designed in order to check the fitness function of an approximate solution [5].

There are many different formal verification tools and algorithms that can be used for fitness evaluation. The ABC synthesis tool [13] is an open source software for synthesis and verification of digital combinational and sequential circuits. It provides efficient implementation of several verification and optimization algorithms as well as various formats and functions for circuit synthesis and manipulation. Because of its open source nature, we employ the existing code base to implement the proposed algorithm.

Contributions

The main contribution of this thesis is the proposal of a method which allows to approximate complex arithmetic circuits with formal guarantees on the approximation error. The method integrates formal techniques employed for approximate equivalence checking into a search-based circuit optimization algorithm based on the Cartesian Genetic Programming. The key idea of our approach is to employ a novel search strategy that drives the search towards promptly verifiable approximate circuits. We evaluated the method's performance on functional approximation of multipliers (with up to 32-bit operands) and adders (with up to 128-bit operands). This is for the first time when such complex approximate circuits with formal error bounds have been presented, which demonstrates an outstanding performance and scalability of our approach compared to the existing methods that have either been applied to the synthesis of 8-bit multipliers or a statistical testing has been used only. Our approach thus significantly improves capabilities of existing methods.

The thesis is organized as follows. Chapter 2 includes basic information about approximation techniques, evolutionary algorithms and describes the chosen method – Cartesian Genetic Programming – in detail. Chapter 3 discusses the formal methods that can be employed for the evaluation of the features and qualities of approximate circuits. In Chapter 4 we propose novel miter constructions and the key feature of this project – verifiability driven

search strategy – that greatly increases the algorithm's performance. Chapter 5 overviews the most important aspects of the implementation of the algorithm in ABC. Chapter 6 is dedicated to experiments and statistical testing of the developed implementation. During the experiments, we constructed a high-quality Pareto set of multipliers up to 32-bit providing trade-offs between the circuit error and size. The best circuits were synthesized into target 45 nm technology and their nonfunctional parameters (such as on-chip area or power consumption) were examined.

Chapter 2

Evolution Based Approximate Circuit Design

Initially, the approximate circuits were designed manually by removing parts of the existing fully functional designs that were not significantly contributing to the result. For example, during the development of approximate adders or multipliers we can save some on-chip area by removing the logic computing the least important bits of the result. This approach is naive and very simple. However, it does not provide interesting results and more complex techniques are needed to achieve further improvements. In this thesis, we focus on the approximate circuit design using evolutionary algorithms which are described in this chapter.

2.1 Approximation techniques

There are three basic approaches to the implementation of approximate circuits [19]:

- over-scaling,
- over-clocking,
- functional approximation.

The over-scaling and over-clocking techniques use ordinary circuits that work perfectly fine under usual circumstances. The first method can reduce power consumption by voltage over-scaling, which can cause the occurrence of occasional errors. Similarly, over-clocking can achieve greater performance by enhancing circuit's working frequency over the maximum frequency at which the circuit still works correctly. This leads to the occurrence of timing errors, but better overall performance.

Unlike the first two methods, functional approximation does not use the original correct circuit but rather a specially created one. This circuit is designed in such a way that it does not fully implement the original logic behavior described in the specification. The simplest method that was already mentioned in the previous section implements functional approximation by omitting the least significant bits of the result and removing related logic. Small approximate circuits can also be designed manually with very good outcome. For example, a two-bit approximate multiplier which was manually constructed consists of 5 gates only and exhibits the delay of 2 d where d is a unit delay of logic gate. Its output is correct for 15 out of 16 possible inputs. A conventional solution requires 8 gates and

exhibits the delay of 3 d. This approximate multiplier has been used in larger multipliers and then employed in image processing applications. Reported power savings are impressive: 30%-50% for a mean error of 1.39% - 3.35% [10].

The manual design of circuits on gate level is only possible for very small circuits. For larger circuits the task becomes unfeasible. Advanced techniques employ the automatic logic synthesis of new approximate circuits whose error must not exceed predefined limit. Measuring the error of the approximate circuit requires expressing its error by one of various metrics such as worst case error, error rate, mean error, etc. Some synthesis tools try to derive the approximate circuit from the original one, other methods use different heuristics and metaheuristics to design the demanded circuit. This thesis focuses mainly on the technique of circuit design called evolutionary algorithms. This metaheuristic, its versions and related topics are described in detail in the following sections of this chapter.

2.2 Evolutionary algorithms

Generally, an evolutionary algorithm is a metaheuristic technique for solving optimization and search problems inspired by the natural selection and evolution of living organisms. It can also be used for the design of both software and hardware systems [12]. The search space consist of all possible solutions, each element of this space represents a specific solution. A computer program, or in our case a logic circuit, is encoded as a set of genes. These genes are then modified using genetic operators to explore the space of possible solutions. Evolutionary algorithms usually work in generations. In the beginning there is a set of initial candidate solutions (first generation), generated either randomly or seeded. The solutions are then evaluated by a metric called fitness function which describes the quality of each solution. A set of the best solutions is chosen and used by the evolutionary algorithm to create a new generation of solutions and the whole process is repeated. This task is iteratively done until a predefined number of generations is reached or a solution with sufficient quality is found.

When applying genetic algorithms to a specific problem, there are two essential tasks to be solved. The first one is to encode the candidate solutions in such a way that allows us to combine and modify them to create new candidate solutions with different features and qualities. Secondly, we have to be able to asses the quality of a candidate solution and decide which solution from a set is the best one. There are many known formats for the representation of different optimization or search problems, some of them are mentioned in the list bellow. The gene representation is sometimes also called a chromosome.

- Bit vectors,
- integer or floating point number vectors,
- graphs,
- trees.

Creating new generation of solutions from the previous ones is performed by the means of genetic operators. There is a great variety of them and their form depends on selected gene representation of the candidate solutions. The most common operators include:

- vector crossover (one point, multipoint, uniform),
- tree crossover,

- mutations.

Crossover (also recombination) usually combines two chosen candidate solutions, called parents, into one or more new solutions, offspring. Mutations choose one candidate solution and perform a slight change on it in order to preserve most of its useful features in the next generations. Typical examples of mutations include flipping a bit value in a bit vector, randomly increasing or decreasing value in integer vectors and swapping or creating nodes in graphs or trees. Examples of these operators performed on 8-bit vectors are illustrated in Figure 2.1

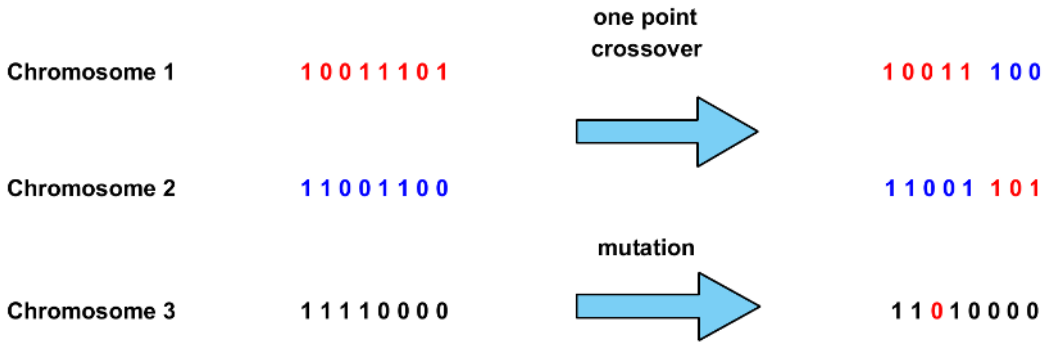

Figure 2.1: Crossover and mutation of binary vector chromosomes.

Evolutionary algorithms are applicable in many different areas of optimization and search problems and its specific form depends heavily on actual application. One alternative that is very useful in designing combination digital circuits on the gate level and thus for the goal of this thesis is called the *Cartesian Genetic Programming* (CGP). This method can be successfully used for the approximate circuit development and is further described in the section below.

2.3 Cartesian genetic programming

Cartesian genetic programming [11] is one variant of the various evolutionary algorithms. It encodes solved problem in a form of oriented acyclic graph. Each node of the graph represents elementary function (logical *and*, *or*, *xor*, etc.) from a predefined set of functions.

Nodes of the graph are connected by oriented edges. Each node has n_a input edges and one output value that can be assigned to its output edges. The whole graph has n_i primary inputs and n_o primary outputs. The nodes of the graph are arranged in a regular two dimensional matrix of fixed size with u rows and v columns. This representation is favorable because it resembles the organization of logic gates in digital circuits and the structure of elements in FPGAs. A generic CGP matrix along with its parameters is shown in Figure 2.2.

Every node's inputs can be connected to either primary inputs or outputs of the nodes in previous columns. Node's outputs can serve as inputs for the nodes in following columns or outputs of the whole graph. Connections between nodes in the same column are forbidden. So called L-back parameter defines the number of previous columns that can be used as inputs for a specific node. For example with the L-back parameter set to 2, a node in the fourth column can use outputs from the second and third columns but cannot be

connected to the outputs of the first column nor the primary inputs. L-back set to 1 allows only the usage of the previous column and is convenient for developing circuits with pipeline processing. L-back set to maximum value (number of columns) enables maximum connectivity and thus the biggest variety of solutions.

Definition of CGP parameters:

- number of rows, u,
- number of columns, v,
- number of inputs, n_i,
- number of outputs, n_o,
- number of each node's inputs, n_a,
- L-back parameter, L,
- set of available functions, F,
- number of available functions, n_f.

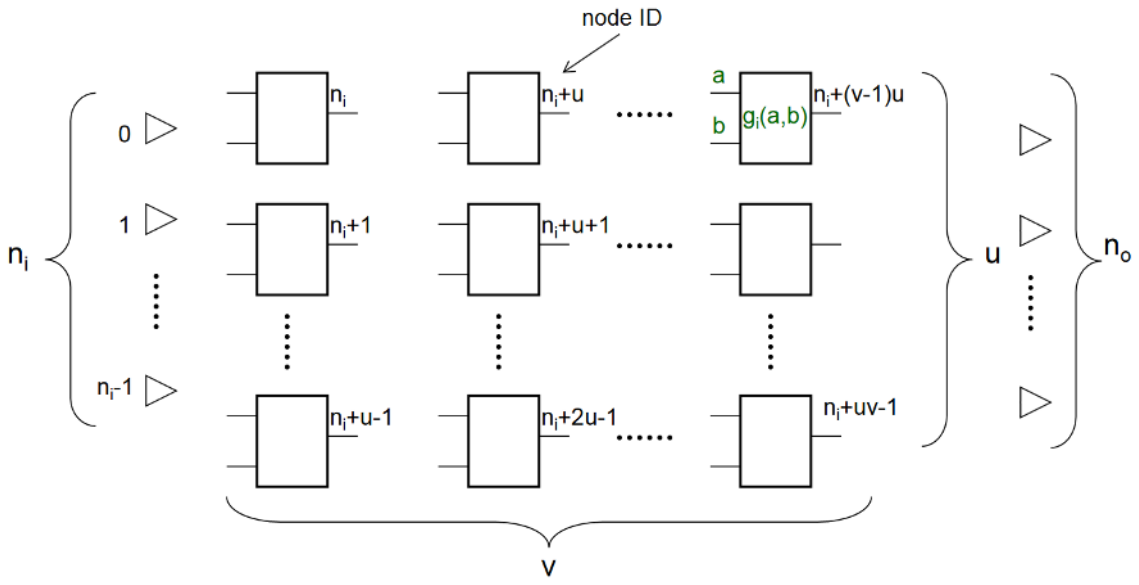

Figure 2.2: Basic scheme of CGP representation.

One of the main advantages of CGP is that it prevents the occurrence of the so called *bloat* phenomenon. During the evolution we usually try to encourage modifications of candidate solutions. When comparing the best solution obtained so far to the new generation of solutions we always prefer the new solution to the old one in case they are equal. This approach allows us to explore areas of the state space of chromosome that would otherwise be unreachable. On the other hand, the selection of new modified candidate solutions also leads to the bloat. As the generations proceed, the chromosome develops ineffective or redundant parts, which are always accepted and passed to further generations. The chromosomes grow larger and larger and their processing slows down significantly. In past, the

size of the chromosome could even exceed the memory capacity of the host computer. CGP does not suffer from the bloat because of fixed chromosome length which is limited by the size of the node matrix. Thanks to this we can perform much more generations of evolution than in the classic genetic programming without running into the bloat problem.

2.3.1 Chromosome representation

Another advantage of Cartesian genetic programming is its simple and compact format of chromosome representation. It can be easily encoded as a fixed length integer vector, as shown in the following example. This representation also allows us an easy manipulation of the chromosome during the creation of new candidate solutions and will be demonstrated on the example in Section 2.3.2.

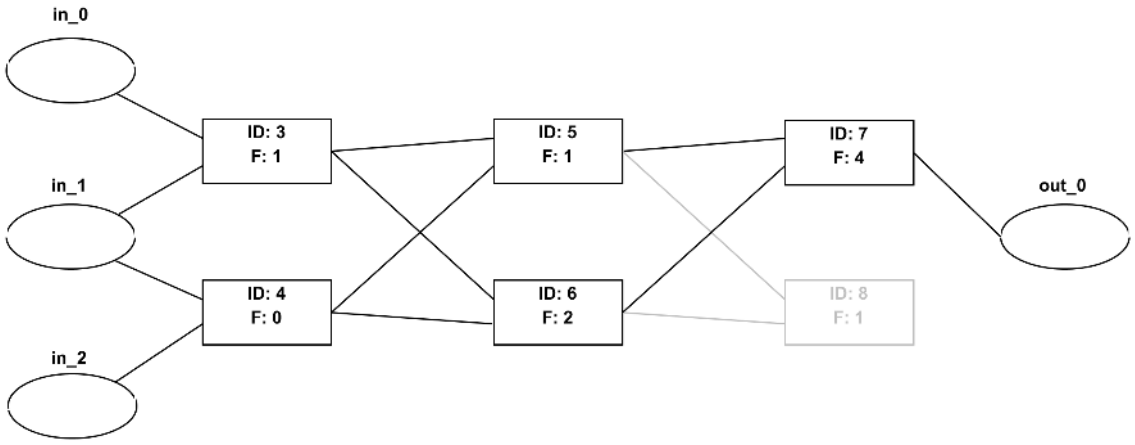

Figure 2.3: Example of CGP circuits representation.

In the Figure 2.3 we can see a simple example of a digital circuit developed using CGP with the following parameters:

- u = 2,
- v = 3,
- n_i = 3,
- n_o = 1,
- n_a = 2,
- L = 1,
- F = [and, or, nand, nor, xor],
- n_f = 5.

The example circuit consists of standard two input logic gates and has three primary inputs and one output. The matrix of nodes has two rows and three columns, six function nodes in total. As we can see, not all of the nodes in matrix necessarily have to contribute to creating the output value. This gives the evolution the space to create various solutions with different qualities (number of nodes, delay, used functions, etc.). The unused node and connections are marked grey in the example picture.

The chromosome representation consist of $u*v$ triplets (u – number of rows, v – number of columns) of integer values, where the first two values entail the identification of the ouputs connected to the first and second inputs of the corresponding element. The third value in the triplet represents the logic function realized by the node. This value is used

as an index into the list of used functions F. At the end of the chromosome there is a tuple of the same size as the number of circuit's primary outputs. This tuple defines, the connections to primary outputs. The beginning of the chromosome can optionally contain a tuple stating the number of primary inputs and outputs, the size of the matrix and other optional information, too. The example circuit is encoded in the chromosome as a list of integers as follows. Triplets for each node are enclosed in brackets for a better readability.

```
(3, 1, 3, 2), (0, 1, 1), (1, 2, 0), (3, 4, 1),
(3, 4, 2), (5, 6, 4), (5, 6, 1), (7)
```

The first four values give basic information about the circuit. Next triplets represent one node of the matrix each. For example, the highlighted triplet has its two inputs connected to the primary inputs 1 and 2. Its function has index 0 and thus the function of the node is logical AND. The last value defines, that the circuit's only primary output is connected to node number 7.

2.3.2 Genetic operators

During the evolution with Cartesian Genetic Programming, only one of the above mentioned genetic operators is used – mutations. The other operators have not brought any interesting results. In each generation, we perform mutations on the best candidate solution found so far. The mutations affect only a small part of chromosome (one or a few integer values). The area of the chromosome to be mutated is usually randomly chosen.

During the mutations, it is vital to respect the given parameters of the circuit – the size of the matrix, L-back parameter and the number of functions available. For instance, when mutating node ID 8 – `(5, 6, 1)` – from the example above, we can only insert values 5 or 6 as its inputs because of L-back parameter set to 1. When changing its function, we can use values 0 to 4 since we have 5 functions available altogether.

2.3.3 CGP pseudocode

The whole process of Cartesian genetic programming can be described by the following algorithm written in pseudocode. In the code we can easily determine the key factors relevant for the algorithm performance. These will be discussed in the next section.

Algorithm 1 Cartesian Genetic Programming pseudocode.

Input: CGP configuration parameters, termination criterion.
1: Randomly generate parent p.
2: Generate P offspring of p using mutations.
3: Evaluate the population.
4: **while** The termination criterion is not satisfied **do**
5: Select the highest scoring offspring b.
6: **if** fitness(b) $\geq$ fitness(p) **then**
7: $p \leftarrow b$
8: Generate P offspring of p using mutations.
9: Evaluate the population.
10: **return** Best candidate found p.

2.4 Fitness function and error metrics

During each generation cycle of Cartesian Genetic Programming (and evolutionary algorithms in general), we need to evaluate the quality of candidate solutions in order to determine which one is the best solution for the problem given. To accomplish this task, we use a fitness function that takes a candidate solution and assigns it a value representing its quality (fitness). The result tells us how closely the specific candidate solution gets to satisfy set aims. In cases where a golden solution of a problem is available, fitness function usually compares the results of a candidate solution to the outputs of the golden solution and the fitness defines, how similar these outputs are. In cases without known golden solution, we need to use a suitable heuristic or approximation. The fitness function can take multiple criteria into account. For example, after achieving a certain precision of the candidate solutions, we can then focus on minimalizing the on-chip area (number of logic gates) of the solution or its logic delay.

The most common fitness function for combination digital circuits, such as adders, multipliers or multiplexors, is the Hamming distance (HD). In this case we evaluate the candidate solution for all possible input combinations and compare its output to the corresponding results of a golden solution (i.e. correct multiplier). The fitness function computing the Hamming distance determines the number of bits where the two compared results differ. Our goal during the evolution is to minimize the value of the fitness. When a solution with zero fitness function is found, we can either end the evolution or focus on other criteria of the solution as was described above.

Hamming distance as the fitness function is simple and useful; approximate arithmetic circuits have different demands, however. HD can be still applied to circuits as multiplexors or decoders, but it does not reflect the qualities of arithmetic circuits (adders, multipliers) well. For example, a candidate multiplier can only have 2% of different bit values in its outputs compared to the correct multiplier, but if the errors occur on the most significant bits, the results do not correspond to the correct outputs at all. That's why we need to employ arithmetic error. This type of error is more demanding in the terms of computational resources, because we need to interpret the output values as numbers and perform arithmetic operations to measure the error. These operations are slower than the bitwise operations for the computations of plain Hamming distance. There are different error metrics we can use to determine the error of an arithmetic circuit:

- total error – sum of absolute values of errors for all input combinations,
- error rate – the number of input combinations that give wrong result,
- worst-case error – the highest error occurring among all input combinations,
- average error.

These metrics can be used to determine the quality of an approximate circuit and hence as the fitness function during CGP evolution. There are various demands on the solutions we can achieve by applying multi-criteria fitness function with the error metrics, i. e. an approximate adder that generates correct output in 95 % of cases, has a minimal average error and covers minimal area on chip, or an approximate multiplier with minimal delay whose error never exceeds a predefined value.

In this thesis, we focus mainly on the worst-case error and the mean average error and define them as follows. For a correctly working circuit G (golden solution) which computes

a function f_G and its approximation C (candidate solution) computing a function f_C, where $f_G, f_C : \{0,1\}^n \to \{0,1\}^m$ (m and n are positive integers denoting the input and output bit widths) are formulated:

$$\mathrm{WCRE}(G,C) = \frac{\max_{x\in\{0,1\}^n} |\mathrm{int}(f_G(x)) - \mathrm{int}(f_C(x))|}{2^m},$$

$$\mathrm{MAE}(G,C) = \frac{\sum_{x\in\{0,1\}^n} |\mathrm{int}(f_G(x)) - \mathrm{int}(f_C(x))|}{2^m},$$

where $\mathrm{int}(x)$ denotes the integer representation of the bit vector x and $|i|$ denotes the absolute value of the integer i.

2.4.1 Fitness evaluation methods

The total time needed to complete a whole evolution cycle can be defined as:

$$T = G * ((T_a + T_c) * P + T_p)$$

- G – number of generations,
- $\mathrm{T_a}$ – time of evaluation of a candidate solution,
- $\mathrm{T_c}$ – time needed to compare fitness values of candidate solutions,
- $\mathrm{T_p}$ – time for the generation of new population,
- P – the size of population.

Generating a new population of candidate solutions by the means of mutation and comparison of fitness values are relatively quick and easy. On the other hand, evaluating the fitness function is very demanding and often makes up most of the computational time. The Cartesian Genetic Programming usually needs a high number of generations to evolve a working circuit (millions of generations). In every generation, we need to assess the quality of every candidate solution. The speed of fitness function evaluation is hence crucial for the successful circuit development.

One of the methods to speed up the fitness evaluation capitalizes on the fact that present processors perform bitwise operations on whole 32 or 64 bit registers. Instead of evaluating one input combination at a time, we can speed up the whole process up to 64 times. In each cycle, we fill the registers with 64 different input values and evaluate fitness for all of these values in parallel. The whole process is demonstrated in Figure 2.4.

Even with this significant acceleration, the speed of fitness evaluation is still not sufficient for designing circuits with a large number of input combinations. One of the solutions to this problem can be performing evaluation of only a small random part of the possible input combinations (random simulation). However, this approach only approximates the candidate's fitness and does not guarantee its exact value. This thesis focuses on a different kind of approach that relies on formal methods to prove the features of a candidate solution. The next chapter aims at these methods and discusses them thoroughly.

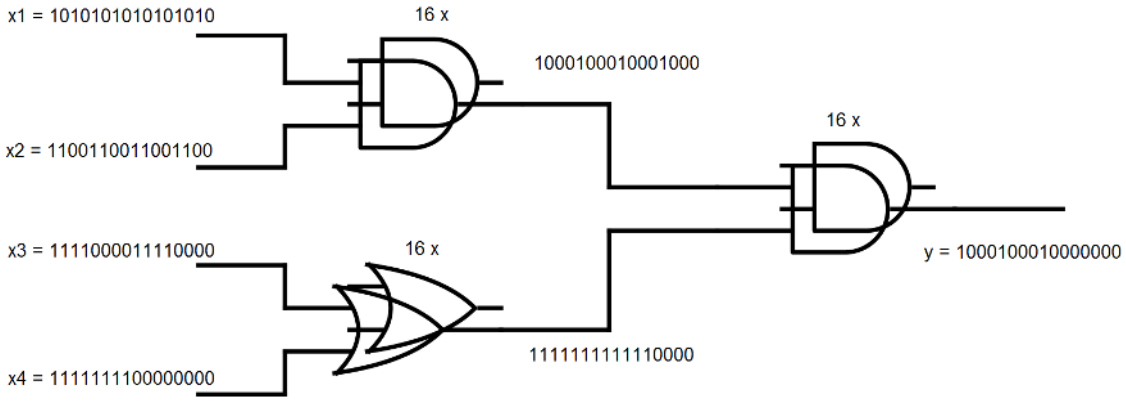

Figure 2.4: An example of parallel fitness evaluation with 4 input variables

Chapter 3

Formal Methods for Fitness Evaluation

Developing larger and more complex circuits with CGP becomes very demanding for three main reasons. Firstly, with growing size and complexity, the evaluation of circuit's outputs takes longer time for every input combination. Secondly, the number of possible input combinations, and thus the time complexity, grows exponentially with the number of inputs. For example, for a 3 bit adder, we need to evaluate 2^6 different input combinations, in case of 4 bit adder, the number of possible inputs grows to 2^8. The third and final reason is that with increasing complexity of the circuit the length of its chromosome increases as well. This means that the state space of the problem to be examined by the evolutionary algorithm grows in size and we need to perform more cycles of the algorithm to successfully explore it.

As we can see, the problem to solve has two major points. We need to execute more evaluations of the candidate solutions, but at the same time each evaluation takes longer. The combination of these problems makes the classic evolution design of circuits with input bit widths bigger than 8 bits unfeasible. In order to achieve satisfactory results, we need to speed up the evaluation of candidate solutions and/or improve the search algorithm.

One of the goals of this thesis is to examine the techniques improving the speed of the fitness computation. Instead of the iterative evaluation of all input combinations, we transform the problem into a formal query describing circuit's features and then try to prove them or find a counter example disproving them. This chapter discusses various formal verification methods that can possibly be used in the previously described approach.

3.1 Basic verification techniques

There are lots of various formal verification techniques. In this thesis, we focus mainly on automated methods that can be potentially utilized for the evaluation of the quality (fitness) of the candidate approximate circuits.

3.1.1 Explicit Model Checking

Model checking [6] (also property checking) is a problem of determining whether a given system meets a defined specification. This is accomplished by a systematic exploration of the state space of the system. The systems in question are usually finite state. The systems are typically hardware or software and the specification contains safety requirements such

as the absence of deadlocks and other critical states that can cause the system to crash. In order to solve this problem automatically, we need to express both the system and the specification in a precise mathematical language. The system to be verified is usually modeled as a finite state machine and the reachable states of this machine are then traversed in order to verify the properties.

The properties are classically specified using temporary logics (LTL, CTL, CTL*, ...), but there are also other forms possible. The properties checked are generally classified as *safety* and *liveness* properties. While the former declares what should not happen (e.g. deadlock), the latter declares what should eventually happen (e.g. system always eventually responds to a message). A counterexample for a safety property is a sequence of states where the last state contradicts the property. A counterexample to liveness properties is a path to a loop of states that does not contain a desired state. Such a loop represents an infinite path that never reaches the specified state.

The advantages of model checking include:

- a high degree of automation,
- easiness of use,
- generality,
- provides counter examples.

On the other hand, the model checking is held back by the problem of the state space explosion. The number of possible states of a specific system grows rapidly as a result of combinatorial considerations. For example, a system with two 32 bit variables can be in $2^{32} * 2^{32}$ different states when only considering these two variables. Another cause of the state explosion is interleaving of concurrent processes – n concurrent processes with m states can generate m^n states! Considering these facts we can see that for almost every practical system the state space gets easily out of proportion and its exploration becomes computationally unfeasible. There are various techniques of dealing with the state space explosion problem, such as:

- efficient storage of state space, symbolic model checking,
- state space reductions (symmetries, partial order reduction),
- abstraction,
- compositional methods,
- bounded model checking.

3.1.2 Symbolic model checking

The *symbolic model checking* [4] made a breakthrough towards wide usage of these techniques. In symbolic model checking, the sets of states are represented implicitly using Boolean functions. For example, we can assume that the behavior of a system is determined by two binary variables v_1 and v_2 and that (11, 10, 01) are the three combinations of values that can be assigned to these variable in any run of the system. Rather than keeping explicit list of these states (as was done in explicit model checking), it is more efficient to handle a Boolean function, that represents this set, i. e. $v_1 \vee v_2$. Manipulating Boolean

functions can be easily done with the application of Reduced Ordered Binary Decision Diagrams [3] (ROBDDs), a compact, canonical graph representation of Boolean functions.

The initial set of states is represented as a ROBDD. The procedure then starts an iterative process where, at each step i, the set of states that can be reached for the first time in i steps from the initial states are added into the ROBDD. At each step, the set of states is intersected with the set of states that satisfies the negation of the property. If the result of the intersection is a non-empty set, an error has been detected. The procedure ends, when the set of newly added states is empty or an error is found. The first case indicates that the property is satisfied, because no reachable states violates it. The latter case generates an counterexample of an incorrect run of the system.

The bottleneck of these methods is the amount of memory that is required for storing and manipulating ROBDDs. The Boolean functions representing the sets of states can grow exponentially. There are many techniques such as abstraction, reduction and decomposition to solve this problem, but the state explosion is still not overcome.

It has been shown that BDDs can be utilized in the design of approximate adders [24] but are not suitable for the development of approximate multipliers. Alternatively, symbolic model checking can be encoded into the boolean satisfiability problem (SAT). This approach is suitable for circuit design and will be described in detail in the next section.

3.2 Verification through satisfiability checking

3.2.1 SAT

Boolean satisfiability problem (SAT) is the problem of determining, whether there exists a variable assignment (interpretation) for a given propositional formula such that the formula evaluates to true. In other words, we ask whether a given Boolean formula can evaluate to true. If such assignment exists, the formula is called satisfiable. On the other hand, if there is no such interpretation, we say that the formula is unsatisfiable.

$$a \wedge \neg b \wedge c \tag{3.1}$$

$$a \wedge \neg a \tag{3.2}$$

Formula 3.1 is satisfiable because there exists the interpretation a = true, b = false, c = true and the whole formula evaluates to true. Formula 3.2 is unsatisfiable because in every possible interpretation the formula always evaluates to false.

SAT is a known NP-complete problem [7]. This means that there is no algorithm that efficiently solves all SAT instances. However, NP-completeness does not exclude the possibility of finding heuristic algorithms that solve interesting practical SAT instances well. These instances come from various areas, such as automatic theorem proving, circuit design, artificial intelligence and software verification. Because of its wide areas of application, SAT has been subject to intensive research and there are many SAT solving tools and algorithms.

Formula representation

Usually, we consider a propositional formula given in the Conjunctive Normal Form (CNF). Such formula consists of conjunction (logic and) of clauses where a clause is a disjunction (logic or) of literals. Literal is a propositional symbol with optional negation. Every propositional formula can be converted to an equisatisfiable formula in CNF using the Tseitin transformation [21].

$$(a \vee \neg c \vee d) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee \neg d) \tag{3.3}$$

In the example Formula 3.3, $(a \vee b \vee \neg c)$ is a clause, $\neg a$ is a literal and a is a propositional symbol. Propositional formulas in CNF are usually represented in a format called DIMACS. This ASCII format is simple for automatic parsing and at the same time readable for humans. Its first line contains information about the number of clauses and the number of variables (propositional symbols) in the form `p cnf nbvar nbclauses`. After that the clauses follow, each one on separate row. Each clause is a sequence of non-zero integer numbers from the interval $< -nbvar, nbvar >$ ending with 0 on the same line. The positive numbers denote corresponding variables, negative numbers mean negation of the corresponding variables. The following example of the DIMACS format shows the encoding of Formula 3.3. The numbers are assigned to the variables in alphabetical order.

```
p cnf 4 3
1 -3 4 0
1 -2 3 0
-1 -4 0
```

SAT solving

Most SAT solving tools are built on variants of the classical Davis-Putnam-Longemann-Loveland (DPLL) procedure. This algorithm starts from the ground CNF formula and tries to build an assignment that satisfies the formula and hence proves that the formula is satisfiable. This assignment is created using a backtracking mechanism.

The DPLL procedure works over an abstract DPLL system. This system is a pair $(S, \Rightarrow)$ where S is a set of states and $\Rightarrow$ is a set of transitions between the states. Each state is a pair denoted $M||F$ where M is an assignment and F is a formula to be verified. There is also a special labeled state *fail*. The procedure starts from the initial state $\emptyset||F$ and builds the assignment using the following, briefly described rules.

- *PureLiteral* – we add literal l to M if F contains l, F does not contain $\neg l$ and l is undefined in M.
- *Decide* – we add literal l^{d} to M if l or $\neg l$ occurs in F and l is undefined in M. Index d denotes that this literal is a decision literal and was put into the assignment by the Decide rule.
- *UnitPropagate* – we add literal l to M if F contains a clause C that has all literals but l defined in M and is still not satisfied.
- *Backtrack* – when the formula F becomes unsatisfiable with the current assignment M, we return to the first decision literal in M, negate its value and remove its decision index. All the literals put into the assignment after the actual decision literal are deleted.
- *Fail* – we enter the fail state if F is unsatisfiable with the assignment M and M contains no decision literal. This means the formula is unsatisfiable.

Opposite to the fail rule, if there is no other rule applicable and the derivation has not got to the fail state, the given formula is satisfiable. The assignment in the final state of the

procedure is an example of satisfying assignment for the formula. The rules are not applied in random order but rather according to a priority scale. Firstly, Backtrack or Fail are applied if applicable. Otherwise, PureLiteral or UnitPropagate are applied if possible. At last, if no other rule is applicable, we use the Decide rule. The motivation of placing Decide to the lowest priority is to reduce the amount of guessing as much as possible. The Decide procedure can be subject to many various and complex heuristic that can significantly affect the performance of the algorithm.

Modern DPLL systems do not use PureLiteral rule in the derivation process but rather perform this task as a preprocessing step. Additionally, a more efficient backtracking algorithm is used. Simple Backtrack method often goes back to irrelevant decision points and then performs the same steps until the solution fails again. This can happen multiple times and leads to useless back and forth computations. Instead, the more efficient BackJump rule analyses the clauses and conflicts in the assignment, skips multiple decision levels and jumps directly to the relevant decision point. These systems can also learn new clauses that do not affect the satisfiability of the formula, but can speed up the algorithm. On the other hand, some unnecessary clauses can be neglected too.

3.2.2 SMT

Satisfiability modulo theories (SMT) [1] problem is a decision problem for logical formulas with respect to background theories expressed in the classical first-order logic with equality. This problem extends SAT by providing more information about the meaning of logical formulas in the supporting theories. In computer science, these theories can typically include the theory of real numbers, theory of integers and the theories of various data structures such as arrays, lists, bit vectors, uninterpreted functions, etc.

SMT instance is a formula in first-order logic, where the function and predicate symbols have additional interpretation and like in SAT, the problem is to determine whether such formula is satisfiable.

There are different approaches to solving SMT. The *eager SMT techniques* take advantage of existing SAT solvers. Firstly, we translate the input logical formula F into a satisfiability-preserving propositional formula F'. Then we can use SAT solver to determine, whether the new formula F' is satisfiable. This approach is not used in practice very often. We need specific translation methods for each theory and the technique might quickly run out of time or memory in many practical applications. There is also the lazy approach with a much better performance.

The *lazy SMT techniques* split the task of the decision procedure into two co-operating parts:

- a propositional SAT solver, which checks the satisfiability of the boolean skeleton of the formula given and views atomic formulas as simple propositional symbols,
- a theory solver, which implements decision procedure for the given theory T.

The first step is to build a dictionary of atomic predicates that appear in the input formula F. We also need to recognize their positive and negative appearances. The first row in the following table lists atomic predicates occurring in Formula 3.4, the second row assigns the propositional symbols for SAT solver to the predicates. Note the positive and negative occurrence of predicate p_1.

$$(a > 1 \vee b < 5) \wedge (a \leq 1 \vee b > 2) \wedge (c \geq 5) \tag{3.4}$$

$a > 1$	$b < 5$	$a \leq 1$	$b > 2$	$c \geq 5$
p_1	p_2	$\neg p_1$	p_3	p_4

The typical lazy SMT decision procedure uses the following interaction of a SAT solver and a T-solver for a single theory involved:

1. SAT solver checks whether the boolean skeleton of the input formula is satisfiable. If it's not, then F is also T-unsatisfiable and the procedure terminates. Otherwise, the procedure continues with the next step.

2. SAT solver finds a satisfying assignment M for the boolean skeleton of F. The theory solver then checks, whether $T(M)$ is satisfiable. If it indeed is, the formula is T-satisfiable and the procedure ends with this result. Otherwise, we cannot yet decide the satisfiability of F and the procedure continues.

3. The theory solver provides a theory lemma which is added to the set of clauses. The SAT solver is then restarted and the whole process begins again from step 1 with the updated set of clauses.

The basic lazy SMT approach has various extensions that improve its performance, memory demands, etc. These extensions include *on-line* SAT solver (after a restart, solver does not begin from empty assignment, but continues from the current one with updated set of clauses) and *incremental* SAT solver (SAT solver can query T-solver during the SAT decision procedure and does not have to wait until a satisfying assignment is found). When dealing with more than one background theory, more complicated approaches are needed, such as the Nelson-Oppen method [17].

3.3 Strict equivalence miters

In order to successfully employ formal verification methods for the evaluation of the fitness function in CGP, we cannot work with the candidate solutions in their chromosome form. We need to transform the representation into logical formulas (propositional or first order for SAT or SMT respectively) or create a formal model for them to be model checked. Moreover, we also need to specify the proper behavior we are trying to achieve.

In case of using a SAT solver, which this thesis aims at, we need to encode the comparison of the candidate solution and the golden solution into a propositional formula. This formula is then examined by the SAT solver which gives us the result, whether or not the candidate solution satisfies the specification. This can be done by connecting the candidate and the golden solution with a special construction called a *miter*.

In our case, the basic miter is a combinational circuit that checks the equivalence of the two solutions in question. These have equivalent behavior if and only if their output values are always the same for any corresponding input values. Comparison of corresponding output bit values can be done easily by connecting them to a newly constructed logical XOR gate. This gate is added to every pair of equivalent outputs of the circuits. After that, to check whether the output of any of the new XOR gates evaluates to the state of logical '1', we perform logical OR on their outputs. Because the number of circuit's outputs is arbitrary and there are usually only two input gates available, we build a tree of OR gates over the XOR gates. The output of the root of the tree indicates, whether the two circuits are equivalent or not.

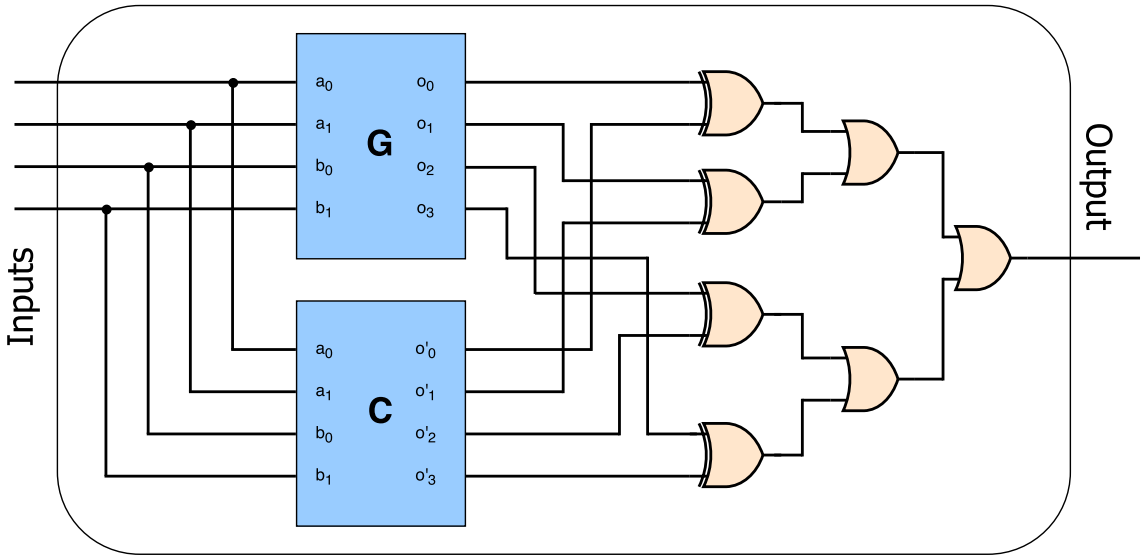

Figure 3.1: Miter checking the equivalence of two circuits.

Figure 3.1 illustrates an example of miter constructed over two 4 bit output circuits. The candidate solution and the golden solution are labeled C and G respectively. XOR operation is performed over every pair of corresponding output bits and their value is then aggregated using OR gates. The comparison of the same input combinations is ensured by the connection of the corresponding input ports to the same input wires.

The whole circuit is combinational and consists only of standard logic gates. Thus, it can be easily converted into a propositional formula. As was already stated above, most SAT solvers work with formulas in the Conjunctive Normal Form (CNF). A general formula can be converted into CNF using the Tseitin transformation [21]. The value of this formula for a given assignment (values of input variables) represents the value of the circuit's single output. In order to solve the equivalency problem, we need to include both the candidate solution and the golden solution into the propositional formula. Moreover, it also has to contain some additional logic for the miter. This results into a rapid growth of the formula and need for an efficient SAT solver. Current solvers are well capable of checking formulas with tens thousand of variables and are able to verify complex combination circuits, such as 16 or 32 bit multipliers.

The SAT solver checking the propositional formula can return one of two possible results – the formula is either satisfiable or unsatisfiable. Unsatisfiability of the formula says, that the output of the circuit always preserves the logic '0' value. This means that none of the XOR gates over the outputs of the golden and candidate solutions ever gets into logic '1' state and thus all the outputs are always equivalent. On the other hand, satisfiability of the formula shows, that the outputs of the examined circuits differ for some input combinations and the circuits are not equivalent.

This miter checks only the strict equivalence of circuits and offers no information about their similarity. That's why it is not suitable for the approximate circuit development. For this task, we need to design different miters that will help us measure the various error metrics described earlier. Section 3.4 focuses on such miters.

3.3.1 ABC

The ABC synthesis tool [13] is an open source software for synthesis and verification of binary combinational and sequential circuits. The main purpose of ABC is to create an academic synthesis tool with its strength equal to common commercial tools. It provides implementations of formal verification and optimization algorithms as well as various formats and functions for circuit synthesis and manipulation. This makes it a suitable tool for the purposes of this thesis. One of the drawbacks of ABC is the absence of documentation, which makes adding new features to the tool challenging. Because of the useful procedures and methods, ABC is suitable for the realization of the algorithm proposed in the next chapter. The algorithm is implemented as one of the modules of the tool and is accessible directly from the ABC command line interface.

ABC provides methods for the basic strict combinational equivalence checking. To examine whether two combinational circuits are equivalent, ABC constructs the previously described miter including the two circuits. The tool then uses one of the inbuilt satisfiability check procedures to examine the satisfiability of the miter's output.

Circuit representation formats

The basic format for circuit representation, manipulation and optimization in ABC is AIG (and-invertor graph). This format represents circuit as an acyclic graph, each node of the graph has the function of logical AND with two inputs. The nodes are connected by edges with arbitrary invertor signs. This basic version of AIG can be extended to sequential AIG by adding another labeling function to the edges marking the count of the registers along an edge. AIG a is very efficient format of representation because of its simplicity, ease of manipulation and the existence of the compact binary AIG format for exporting the circuits.

Apart from AIGs, ABC supports other representation formats, such as netlist, logical network, binary decision diagrams and technology mapped network (FPGA look up tables, a set of gates). ABC can also read input files in various formats (Verilog, BLIF, PLA, EDIF, BAF, truth table, etc.).

Implemented tools, functions and algorithms

The functions implemented in ABC can be divided into following categories.

- Input and output – reading and writing different circuit representation formats.
- Combinational synthesis – starts with an AIG representation or SOP network. Then can perform optional optimizations using rewriting, renoding and redundancy removal. Finally performs the logical synthesis into simple netlist consisting of logical gates defined in cell library.
- Sequential synthesis – similar to combinational synthesis, using also registers and latches. Can also perform retiming to achieve better performance of pipelined circuits.
- Technology mapping – look up table mapping into FPGAs or standard cell mapping.
- Verification commands – can perform strict equivalency check of two circuits or build a miter over them. Also features SAT solver implementation and thus can examine the satisfiability of a circuit's output. This area of functions does not include any relaxed equivalency checks, these have to be further implemented.

3.4 Approximate equivalence miters

The simple miter shown in the previous section is not suitable for our purposes because it only checks the strict equivalence. In order to compare two circuits with regards to a chosen error metric, we need to rely on more complex miters. Their design, implementation and comparison is one of the aims of this thesis. In this section, we provide an overview of existing miters, their features and utilization.

3.4.1 N-th bit error

The simplest miter checking the degree of two circuits equivalence can be constructed by checking the equivalence of each corresponding bit of their outputs separately. We start from the most significant bit, create the miter and perform the check using a SAT solver. If the formula is not satisfiable (the outputs are the same for all possible input combinations), we iteratively go downwards to the least significant bit. The procedure is stopped at the first output bit that is not equivalent. This is the most significant bit of the candidate solution that differs from the golden solution. The index of this bit in the result of the circuit is i, assuming that the least significant bit has index 0. The maximal arithmetic error of the candidate solution therefore does not exceed 2^{i+1}. The example of such a miter is shown in Figure 3.2.

This information, however, does not say much about the real value of arithmetic error, it only defines its maximum limit. For example, consider following situation. We compare two circuits with 4 bit outputs. The expected output for an input combination is '1000' and the candidate solution gives '0111' for the mentioned combination. Comparison via this miter detects different values on the most significant bit and claims that the error does not exceed $2^4 = 16$. This is the worst case and the candidate solution is labeled as unsatisfactory, even though the real arithmetic error is 1 and hence the approximation is very good. This technique does not take the arithmetic nature of the outputs into account much and can easily reject appropriate candidate solutions.

The output of the checked circuit is constructed using only the two output bits in question. The other outputs are omitted. The parts of the circuits that do not participate in creating the result of the circuit are removed during an optimization process and this way, a significant part of the logic is eliminated. The resulting propositional formula for SAT solver is therefore smaller and can be solved much faster. Still, this method does not seem to be applicable as the fitness function in evolution algorithms without further improvements.

3.4.2 N-th bit error with counterexamples

The first miter mentioned in this section not only often discards good candidate solutions but it also cannot decide, which solution is the best from a set of solutions with the same most significant erroneous bit. To solve this problem, we need to examine the counterexamples SAT solver returns. After evaluation of the candidate solution's outputs for the satisfying assignment found by the solver, the exact arithmetic error can be determined. This procedure has to be repeated for every existing counterexample in order to find the maximal arithmetic error.

This technique can help to determine which candidate solution is better from a couple or a set of similar ones. On the other hand, finding all the counterexamples with the SAT solver could be very time consuming, because there could be a great number of them. Moreover,

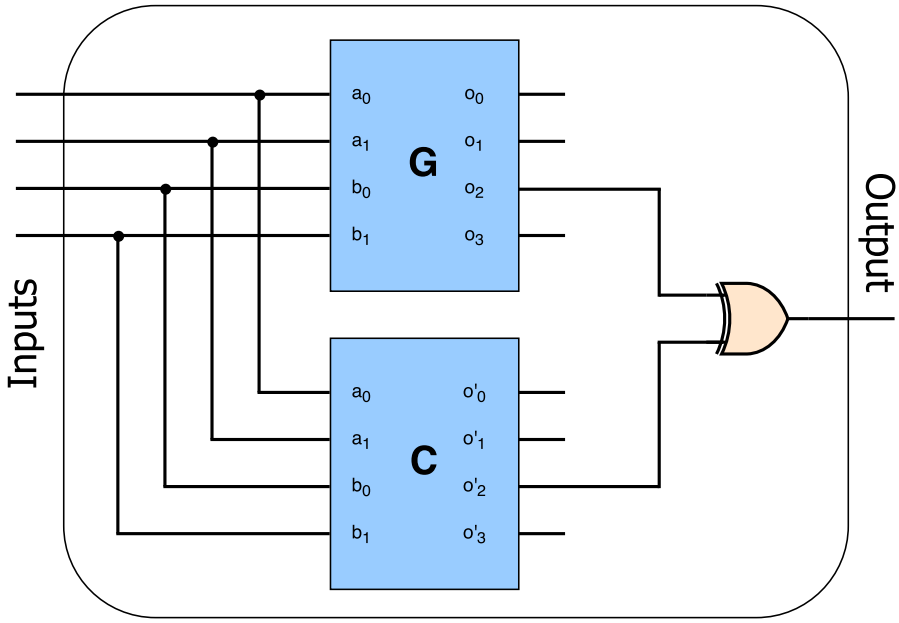

Figure 3.2: Miter checking the equivalence of the third output bit of two circuits.

we also need to evaluate the outputs of the candidate solution for all the counterexamples and would be performing analogical task to the classical fitness evaluation mentioned in Chapter 2. This would likely lead to a similar performance and is a task we try to avoid.

3.4.3 Miter with subtractor and absolute value result

To determine the real maximal arithmetic error, not just the maximal possible one, a more complex miter is needed. To compute the error, the miter over the checked circuits is extended to contain an arithmetic subtractor. This new element subtracts the results of candidate solution from the results of the golden solution. In the previous projects and articles, a subtractor with absolute value results was usually used. Such circuit is constructed from an ordinary subtractor connected to another subtractor circuit which performs the absolute value conversion. The advantage of this approach is the fact that the arithmetic error on the circuit's outputs is always a positive value. On the other hand, such circuit is larger than a simple subtractor giving results in two's complement and therefore slows down the miter evaluation.

In order to determine the maximal error, we check the satisfiability of the subtractor's outputs. The procedure is the same as mentioned in the section about the n-th bit miter. Satisfiability of bit index i in the outputs of the subtractor means that the arithmetic error of the candidate solution is located in the interval $< 2^i, 2^{i+1} >$.

Application of this miter makes the checked circuit much more complex. For multipliers, the subtractor's inputs have double the bit width of candidate solution's inputs. There are also long carry chains inside the subtractor and on the whole, the resulting propositional formula representing the checked circuit will be much longer. This means longer fitness evaluation and deterioration of algorithms performance.

However, during the evolution we usually do not need to determine the exact error of a candidate solution. The knowledge, whether the maximal error of a solution is worse than the error of the best solution found so far, is enough to decide whether the current solution

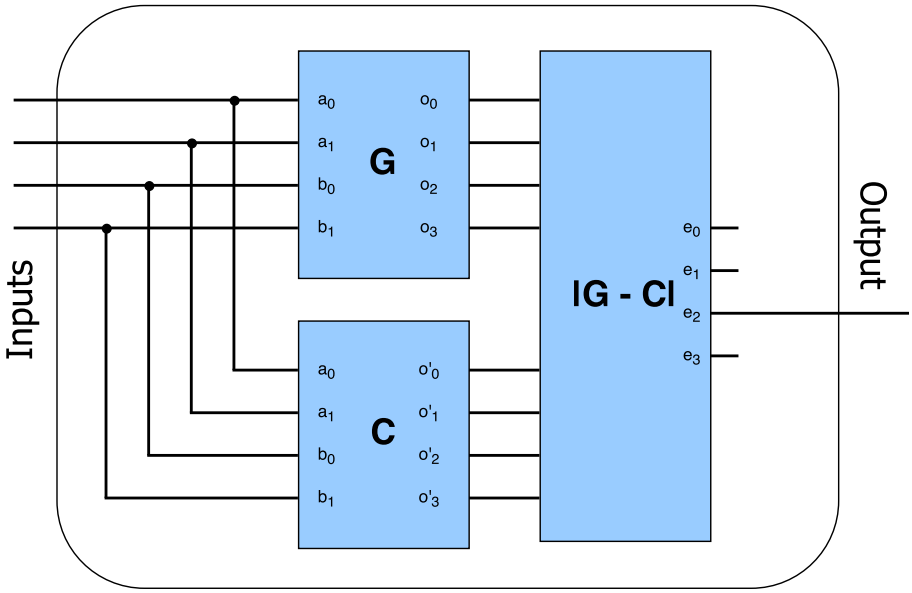

Figure 3.3: Miter with subtractor and absolute value output.

is inferior to the best one. In this case, the worse solution is not further examined and the algorithm keeps the best solution for the next generations.

Let's assume that the best solution found so far has the maximal error on the fourth bit. This solution is used to create a new population of candidate solutions with the help of the mutation genetic operator. Each of these new solutions can be checked whether it has an error on the higher bits by the miter shown in Figure 3.3. If the output of the miter is satisfiable, we know that the solution checked is inferior to the best one and can be deleted. In case the output of the miter is not satisfiable, a more detailed check of the solution is needed, i. e. check of the counterexamples with the exact error evaluation or a more complex miter construction.

The miters illustrated in Figure 3.3 and 3.4 can only be used to asses whether the worst case error is lesser or greater-equal than 2^i where i is a positive integer denoting the lowest index of the examined subtractor outputs. In order to perform the satisfiability check for arbitrary values, we need to construct an comparator circuit over the subtractor outputs.

With the application of this miter, the comparison of two candidate solutions can be often determined in one run of the SAT solver. The success of this technique depends on the assumption that most mutations during the evolution spoil the solution and only a small amount of generated solutions achieve an improvement. The solver can detect the satisfiability of the miter in most cases quite easily using random simulation and other methods. Only after these methods fail to find a counterexample, a lengthy procedure that proves the unsatisfiability of the whole miter is employed.

3.4.4 Sequential miters

Combinational miters are not sufficient to measure more complex error metrics, such as error rate or total error. To determine these, we need to design miters with memory – sequential miters. Such miters are proposed in article [5]. The authors of the article claim, that such miter can be used to assess candidate solution's worst-case error, error rate,

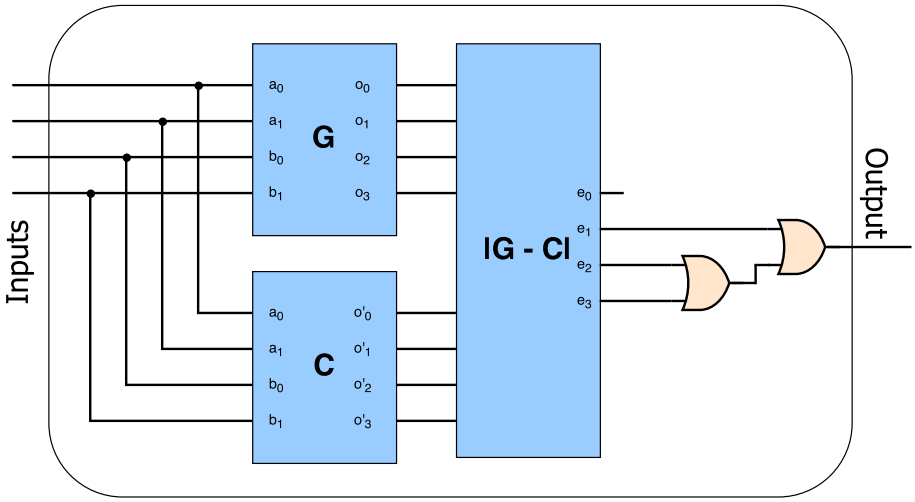

Figure 3.4: Determining whether the circuit has error on the highest three bits of subtractor outputs.

average error and total Hamming Distance. Its basic structure is shown in Figure 3.5. The following list describes its components.

- Gen – generator, creates the input values for the circuits being compared.
- C, G – candidate solution and golden solution.
- E – block performing error evaluation.
- A – accumulator, a block with memory element used to store and accumulate the values of the error metric.
- D – decision block, determines whether or not a predefined condition was violated.

The whole scheme is designed to be universal. The functional blocks can have different assigned function or condition in order to compute various error metrics. Such miters are already quite complex structures and cannot be easily transformed to basic SAT problem because of their sequential nature. The performance of this method has not been evaluated in literature yet and thus will be under further scientific research and examination.

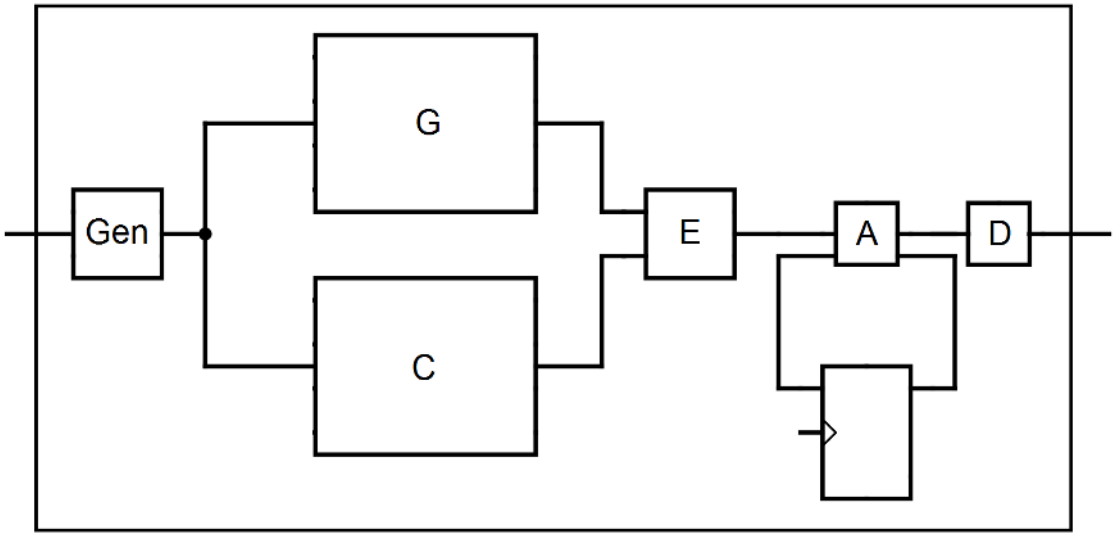

Figure 3.5: The general scheme of sequential miter.

Chapter 4

Scalable Approximate Circuit Design

In the first part of this chapter, we propose a novel miter designed to promptly evaluate the $WCRE$ of a candidate circuit. This miter preserves the useful features of the existing miters described in the previous chapter and at the same time outperforms the other miters in the aspect of circuit area, which leads to a significant performance improvement. In the second part of the chapter, we present they key idea of this thesis – verifiability driven search strategy. This approach allows us to design approximate circuits of unpreceded bit widths – up to 32-bit multipliers and 128-bit adders.

4.1 Miter with subtractor and two's complement result

In order to achieve better fitness evaluation performance, we look to further improve the miter proposed in the previous chapter while preserving its valuable features. The part of the subtractor making the absolute value conversion forms a significant portion of the miter's size. It is possible to simplify the miter and save a number of logic gates by completely omitting this part. However, now we need to consider both positive and negative values of the computed arithmetic error separately.

To achieve this, we add a complementary part to the logic gates presented in Figure 3.4. The new logic in Figure 4.1 examines the negative values of the arithmetic error in two's complement. We can see that it is necessary to detect logical '0' on the subtractor outputs to respect the format of negative values in two's complement. Additionally, the resulting bit flags for positive and negative values are ANDed with the negation of the sign bit and the straight value of the sign bit respectively to only detect values that are truly positive resp. negative and avoid false positive error detection. Finally the outputs of the AND gates are ORed to form the circuit's single output.

In Figure 4.1, we can notice that the subtractor now has 5 output bits instead of the original 4. The extra bit is reserved for the sign of the error value and is employed in determining, whether the error is positive or negative as described above. The formulas defining error metrics in Section 2.4 remain unchanged, note that the $(m+1)$-th bit of the difference is used as the sign bit. At closer exploration, we can see that this naive miter does not even detect the same range of errors for positive and negative values. The miter is unsatisfiable for error in interval $< 1, -2 >$ and satisfiable for other error values. Therefore, such symmetric miter construction is not suitable for our purposes.

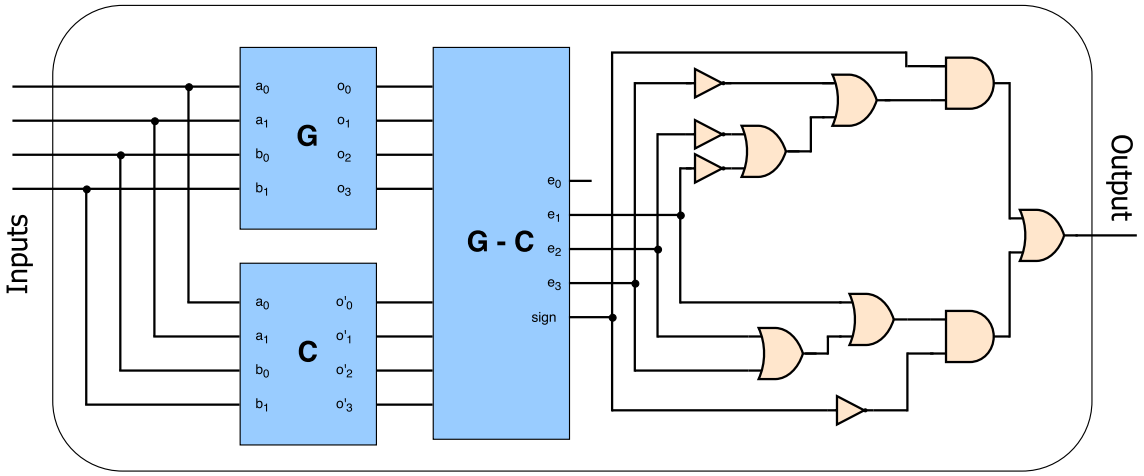

Figure 4.1: Determining whether the circuit has error on the highest three bits of subtractor outputs without absolute value computation.

We can see that a miter built this way contains some additional logic, but there are decent overall savings because the absolute value evaluation is avoided. The exact area savings of miter construction method without absolute value computation gained from conducted experiments are included in Section 5.3.

4.2 Miter with subtractor and comparator

The previous miter constructions already feature some beneficial qualities, however, they are still limited to the powers of 2 and lack the possibility to check, whether the maximal arithmetic error does not exceed a given arbitrary threshold value T. To solve this issue we add a comparator circuit to the miter described in Section 4.1 with inputs A and B, where A corresponds to the m least significant bits of the output of the subtractor (arithmetic error) and B represents the threshold error value T to be examined. The $(m+1)$-th bit of the subtractor is used later to determine whether the error is a positive or a negative value. Such a circuit can be described by the following formula:

$$A > B \equiv \bigvee_{i=0}^{m-1} \left(A_i \wedge \neg B_i \bigwedge_{j=i+1}^{m-1} \overline{A_j \oplus B_j} \right),$$

Since variable B is fixed to the constant threshold value T, we can simplify the comparator formula by propagating the constant during its construction. With this approach we may remove the emerging long carry chains of XOR gates that can have negative impact on the speed of the miter evaluation. The new comparator procedure is denoted by the formula:

$$A > B \equiv \bigvee_{0 \leq i \leq m-1 \ \wedge \ B_i = 0} \left(A_i \bigwedge_{i < j \leq m-1 \ \wedge \ B_j = 1} A_j \right).$$

We can see that the improved formula completely avoids the use of XOR gates. In order to construct the comparator with the minimal number of logical gates, we propose

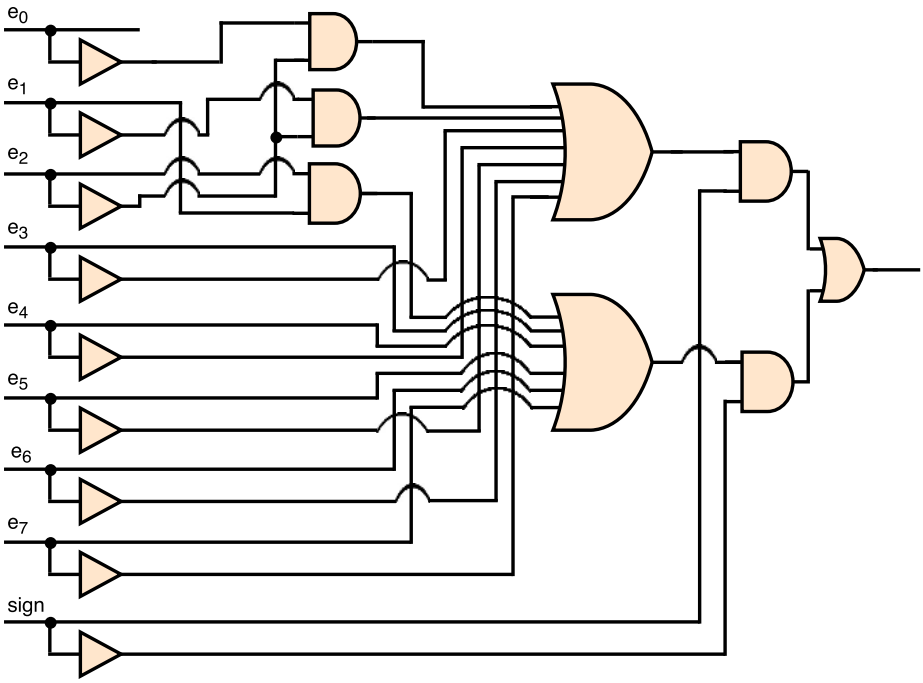

Figure 4.2: Miter constructed with the use of propositional formula describing the comparison of the error value and a constant.

an algorithm that sequentially examines the bit values of threshold T and constructs the comparator over the subtractor's outputs. An example miter circuit for an 8-bit multiplier and threshold $T = 5$ is illustrated in Figure 4.2. The proposed algorithm is outlined in pseudo code in Section 5.3

This miter offers the best features we were able to design in the scope of combinational miters. It allows us to check whether the candidate circuit's C error exceeds a given arbitrary error threshold T. Thus, the miter described in this section is used in the final implementation of the proposed method.

4.3 Verifiability-driven search strategy

We propose the verifiability-driven search strategy as a general concept improving the scalability of the evolutionary design techniques. We demonstrate its key idea on the following problem, but it can be naturally used also in different settings.

Problem 1: *For a given golden circuit G and a threshold $\mathcal{T}$ our goal is to find the circuit C^* with the minimal size such that $WCRE(G, C^*) \leq \mathcal{T}$.*

This problem formulation allows us to reduce the time needed for computing the fitness function f by decomposing the circuit evaluation into two parts. In the first part, we try to prove that a candidate model satisfies $WCRE(G, C) \leq \mathcal{T}$. If the proof is successful, $f(C)$ is defined as the size of C. We use miters and the satisfiability check procedure to decide whether $WCRE(G, C) \leq \mathcal{T}$. Since this decision procedure still represents the most time consuming part of the design loop, we avoid calling the procedure as much as possible. In particular, we call the procedure only for candidates C, where size of C is smaller than the size of the best solution B with the acceptable error (i.e. $WCRE(G, B) \leq \mathcal{T}$) we have found

Algorithm 2 Verifiability-driven search scheme

Input: Golden circuit G, threshold $\mathcal{T}$, resource limit R and time limit T

```
1: B ← G
2: repeat
3:     Generate a population P from B using mutations
4:     for all C ∈ P do
5:         if Size(C) ≤ Size(B) then
6:             if Verifier(G, C, T, R) = UNSAT then B ← C
7: until RunTime() ≤ T
8: return Best candidate found B
```

so far. This concept is illustrated in the diagram in Figure 4.3. Conducted experiments have shown that most of the candidate solutions trigger the first or the second condition in the evaluation chain and their fitness is easily assessed without the decision procedure query. Moreover, there is also no need to build the miter construction, a fact that leads to another performance improvement.

Our experiments indicate that a long sequence of candidate circuits B_i improving the size and having the acceptable error has to be typically explored to obtain a solution that is sufficiently close to C^*. Therefore, both the SAT and the UNSAT calls to the decision procedure have to be short. To this end, we use an additional criterion for the circuit evaluation, namely, the ability of the decision procedure to prove that the candidate circuit has the acceptable error with given resource limits R (maximum number of conflicts at a single node of the network before the verification ends with failure, this constraint also roughly limits the maximum time spent in the procedure). Such a criterion drives the search towards candidates, that can be promptly evaluated.

In our verifiability-driven search scheme, we use the limit criterion in a new way. We intentionally leave out improving candidates B_i that require a longer, but still feasible, verification time. By mutating such candidates we would most likely obtain solutions that would require the same or even longer verification time. Hence finding the whole improving sequence would become computationally unfeasible. Instead, we require that every improving candidate B_i has to be verifiable within a certain limit of SAT solver resources (maximum number of conflicts at a single node) and thus drive the search towards candidates B_i that for a given limit on the overall design process lead to longer improving sequences. These sequences can indeed result in candidate circuits that are closer to C^*. Since we are able to evaluate a much larger set of candidate circuits, we have a better chance to find within the given time a long improving sequence, provided that it exists for the resource limit on a single verification step.

The obvious disadvantage is that we possibly cut improving sequences that would lead to good solutions within the given design time. This disadvantage is even more critical if no sequence for the verification resource amount exists, but there exists a sequence for a slightly larger resources (e.g. two times bigger). Despite this limitation, the results of our extensive experimental evaluation clearly show that the proposed verifiability-driven search allows us to utilise the given design time in a more efficient way comparing to the standard evolution schemes.

Algorithm 2 illustrates the verifiability-driven search used in our approach. It is very similar to a general CGP, namely we use mutations to generate a population P from the candidate circuit B representing the best approximation we have found so far. Note that we start the approximation process with the golden model which is in the agreement with idea to use promptly verifiable improving candidates B_i. The key difference lies in the candidate

evaluation, namely in the procedure $\mathsf{Verifier}(G, C, \mathcal{T}, R)$ returning SAT if and only if we can prove that $WCRE(G, C) \leq \mathcal{T}$ within R solver resources.

4.4 Improved evolutionary loop

Cartesian Genetic Programming works with a population of candidate solutions, in each generation the candidates are evaluated and the best candidate (parent) is used to create the next generation of solutions (parent is included) using mutations. The initial solutions available at the start of the algorithm can be either randomly generated or seeded. The random initialization is successful only for relatively simple circuits (3-bit multipliers, 4-bit adders) but has the potential to generate better final solutions. This approach was tested within our method as well, with the algorithm firstly trying to minimize the candidate's error to be lesser than the error threshold T (feasible solutions) and only after achieving such a solution trying to optimize its size. However, the algorithm does not converge to acceptable solutions well for two main reasons: a) the state space to explore is too large (8-bit multiplier consists of roughly 350 gates, 16-bit multiplier of 1500 gates) and b) the granularity of the first fitness function (candidate's worst case error) is not sufficient to distinguish between two seemingly equal candidates (we can determine, that some candidates have $WCE = X$, but cannot quickly identify the one, that gives this error for least input combinations). Because of this, we use the second approach and seed the initial solution S with a correctly working (golden) circuit or an approximate version of the circuit that meets the constraint $WCE(S) < T$. This circuit is gradually altered by performed mutations and the algorithm tries to minimize its size (fitness function) while maintaining the $WCE < T$ constraint.

For better transparency, we include an illustration of the modified CGP algorithm and summarize its function in Figure 4.3. The further differences between the original CGP code and our modification are described in this section. Note that the algorithm can use the golden solution both as the initial seeding solution and the referential solution for the relaxed equivalence checking. We can see that the candidate's error enters the evolutionary loop only as a *hard* constraint – solutions not satisfying this constraint are immediately discarded. The algorithm performs the cycle until the computational time is depleted. Eventually, the best solution (candidate with minimal size) that satisfies the error threshold T is returned.

The quality of a candidate solution is evaluated in three different stages as shown in the Figure 4.3. As was already stated above, the most time consuming part of the evaluation process is the check whether $WCRE$ of the candidate solution exceeds threshold T. In order to reduce the number of the calls of the decision procedure to minimum, we employ two major optimizations.

4.4.1 Mutations in active string detection

The first one is based on the redundancy of the chromosome representation. As the candidate solutions grow smaller during the evolutionary search, some of the logical gates represented by the chromosome cease to participate in the output computation (are not connected to the primary outputs), but the chromosome length remains fixed. Each of the chromosome gates can be marked as either active or inactive, by a simple chromosome traversal from the primary outputs to the inputs. During the mutations stage, we can examine the parts of the chromosome which are altered and determine, whether the active

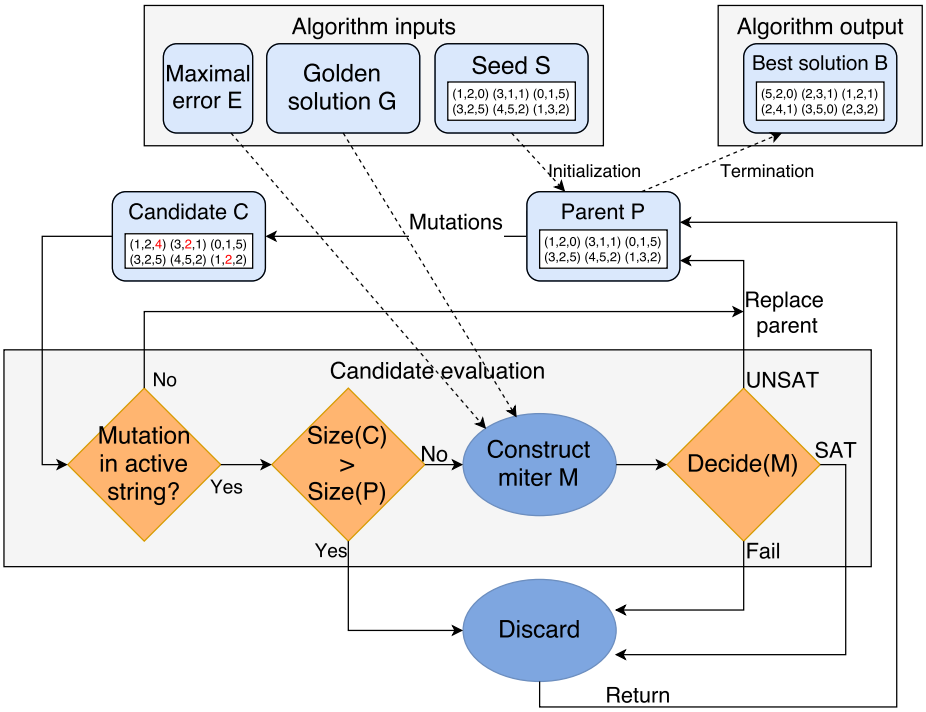

Figure 4.3: Overview of the evolutionary loop with the proposed optimizations and algorithm inputs and outputs.

part of the chromosome was changed. In case only the inactive gates of the chromosome were subject to the mutations, we can say that the individual has the exactly same qualities as its parent without any further evaluation. Because neutral mutations such as this one are supported by the CGP algorithm, we set the individual as the new parent (best solution found so far). Experiments show that during long evolutionary runs with significant savings of the circuit's area (e.g. the candidate has 20 % of the original circuits area), a significant amount of candidate solutions is created using neutral mutations only and the evaluation of such candidates can be skipped. This fact results in a great performance improvement as neither the area nor the $WCRE$ of such individuals have to be examined.

4.4.2 Candidate area approximation

The second optimization also allows us to avoid unnecessary $WCRE$ checks and is based on the fact that the candidate's area can be evaluated much faster than its error. Before evaluating individual's $WCRE$ using the miter and satisfiability decision procedure, we examine its area and if the area is greater than the parent's area, we can immediately discard such a candidate without further examination.

To precisely determine the area that a candidate solution would require in the target technology would require the employment of a hardware synthesis during each generation of the evolutionary cycle. The synthesis of a circuit into the target technology is a computationally demanding and time consuming process. Instead, we can try to only approximate the candidate's area with the information available from the chromosome representation. The simplest approach is to mark the active gates with the same procedure that detects

neutral mutations. The total amount of active gates in the candidate's chromosome can serve as a basic area approximation.

Gate	**Area** [μm^2]
INV	1.4079
AND, OR, NOR, BUF	2.3465
XOR, XNOR	4.6930
BUF	1.8772

Table 4.1: Estimated area of logical gates in 45 nm technology.

This approximation can be further improved without making it significantly more complex. To better estimate the area, user can specify a *liberty* file in the CGP configration. The liberty file contains the estimations of area for each gate of the target technology. Table 4.1 contains an example of logical gate areas for 45 nm technology used during the experiments conducted in this project. We can see that the size of various gates can differ significantly. The connection between the size approximation and the final size after circuit synthesis is shown in Chapter 6.

Chapter 5

Implementation in ABC

ABC synthesis and verification tool (briefly described in Section 3.3.1) was chosen as a base for our implementation, because it is both open source and contains the state of the art methods and algorithms for circuit synthesis and verification. Although it lacks detailed documentation, its source codes are well-arranged and the tool's interface allows for an easy circuit construction and manipulation. The method proposed in this thesis is implemented as a module integrated within ABC and can be used directly from the tool's command line interface. Without the utilization of ABC and its circuit synthesis functions, the amount of work needed to implement such a program would be well beyond the scope of this thesis.

As was already stated above, the ABC tool provides means to construct a miter performing the strict equivalence check over two circuits. The functions used to build this miter can be altered to produce different miters (such as the ones described in Chapter 3.4). The resulting circuit is then checked by the decision procedure *iprove* available in ABC. During the initial stage of this thesis, the simple miter checking N-th bit equivalence (Section 3.4.1) was implemented in the ABC tool and its performance was examined.

Its function was verified using both approximate and correct 16-bit and 32-bit multipliers. In case of structurally different 32-bit circuits, the equivalence check of N-th bit was performed in roughly 0.63 seconds after the miter optimization and in 1.4 seconds without miter optimization in case of equivalent output bits (decision procedure calls resulting in UNSAT result). Output bits that can differ were proven different in the matter of milliseconds (calls resulting in SAT). Since other verification tools struggle to verify even 16-bit multipliers, these results look very promising and make ABC a suitable option for the proposed method realization. The algorithms and features described in the previous Section are implemented in ABC to attain a high performance approximate circuit design tool.

5.1 Circuit representation formats

The algorithm uses circuits represented in several different formats. Each of the formats has its own advantages and disadvantages and is used during different stages of the implemented algorithm. For instance, the AIG format is utilized during the miter satisfiability check. However, it is not suitable to the quick and easy application of the mutation operator and does not feature redundancy like the chromosome format. Therefore, procedures quickly converting circuit representations between the formats have been implemented within the tool as well. The following list summarizes the circuit representation formats used within the project and briefly describes their utilization.

- Verilog / BLIF:
  - the input of the golden solution,
  - the input of the subtractor circuit,
  - the output of the best solution found.
- Chromosome representation:
  - the input of the seeding solution,
  - the internal representation for mutations and fitness evaluation,
  - the output of the best solution found.
- And Inverter Graph (AIG):
  - internal representation of golden solution and subtractor,
  - miter representation,
  - decision procedure input.

5.2 CGP configuration

Cartesian Genetic Programming has many various parameters (mentioned in Section 2.3) that have to be defined prior to the algorithm start. Their assignment through command line arguments would be very complicated and tedious. Instead, we decided to use a configuration file as a single parameter. The configuration file contains definitions of the file paths and parameters necessary for successful evolutionary algorithm execution. An example of such a file is included in Appendix A. The following list briefly describes the parameters of the configuration file:

- Generations – maximal number of generations performed.
- Runs – number of independent runs of CGP executed.
- Max. relative error – maximal $WCRE$.
- M and N – number of columns and rows in the CGP grid.
- Lback – level back connectivity.
- Inputs and Outputs – number of circuit's primary inputs and outputs.
- Population max. – maximal number of candidates in population.
- Mutations max. – maximal number of mutations performed between a parent and his offspring.
- Gates used – number of logical gates types used during the evolution.
- Max. alg. time – time limit for the whole algorithm execution.
- Max. run time – time limit for each CGP run.
- Golden, seed, subtractor and log file names and paths.

- Liberty file with the size of logical gates in the target technology.
- Output module name and file path for the final circuit output.

The `Gates used` parameter determines, how many logical gates from the gate list are available to mutations during the evolution. The list contains basic one and two input logical gates in the following order: AND, OR, XOR, NAND, NOR, XNOR, NOT1 (negation of the first input), NOT2 (negation of the second input), BUF1 (identity of the first input). For instance, usage of `Gates used = 3` would restrict the mutations to only AND, OR and XOR gates. The evolution is usually allowed to use the whole set of logical gates in order to have the maximal search space available to explore.

At the beginning of the algorithm, all necessary parameters are read from the the configuration files and the input circuits loaded from the locations specified in the same file. The initial generation consist of the candidate with the chromosome representation of the seeding circuit and $Population\ max. - 1$ of its offspring. The chromosomes of the offspring are altered using the mutation operator. The number of mutations performed on each offspring is in interval $< 1, Mutation\ max. >$. Literature suggests that about 3 % of the chromosome should be altered between the generations. For an 8-bit multiplier consisting of roughly 350 gates, the $Mutation\ max.$ parameter should be set to approximately 15. The mutations are performed with respect to the definition of the chromosome representation (e.g. gate's input must not be connected to gates in the same or next column, etc.).

5.3 Miter construction

In case the swift optimization procedures described in Section 4.4 do not resolve the candidate's quality without error evaluation, we need to construct the miter and examine whether the candidate's WCE does not exceed threshold T. The circuits needed to build the miter – golden solution and subtractor – are loaded at the start of the algorithm from their source files. These circuits are then optimized using the advanced optimization procedures available in ABC and their AIG representation is stored.

During the construction of the miter, the golden solution and the candidate are merged into one circuit and their corresponding inputs are connected to the circuit's primary inputs. The subtractor is then connected to their outputs to correctly compute the difference between the golden and candidate solutions' results. Finally, the comparator circuit executing the comparison of the subtractor's output to WCE threshold T is constructed. The comparator has a single output which is connected to the primary output of the whole miter circuit.

In order to efficiently build the comparator with the least possible number of logical gates, we propose a new algorithm described in pseudo code in Algorithm 3. The code example performs the construction of comparator for the positive values of threshold T. The comparator's output acquires the logical '1' value if and only if $WCE > T$. Similarly, the comparator for negative values is constructed and their outputs are joined in a logical OR gate to form the single bit output.

The function of the proposed algorithm will be demonstrated on a construction of comparator for threshold $T = 5$ a binary value of 00000101. In the process of the construction, we examine the threshold value iteratively from the least to the most significant bit and build appropriate gates on the corresponding subtractor outputs at the same indexes. Firstly, we skip all the '1' occurrences until the first '0' bit. Then we build an AND gate upon

Algorithm 3 Improved algorithm for comparator construction (positive numbers)

Input: Threshold $\mathcal{T}$, number of subtractor outputs m, subtractor outputs outs[m+1]

```
currentGate = NULL; first = 1;
for (i = 1, iter = 0; iter < m; i <<= 1, iter++) do
    if !(i & T) then
        if first then
            first = 0;
            currentGate = outs[iter];
        else
            currentGate = OR( currentGate, outs[iter] );
    else
        if !first then
            currentGate = AND( currentGate, outs[iter] );
if currentGate != NULL then
    currentGate = outs[m];
return currentGate
```

the current output and the output of the previous gate for every '1' bit occurrence and similarly an OR gate for every '0' bit occurrence. Since the construction examines positive values only, the output of the final gate is ANDed with the negation of the sign bit. The construction for negative numbers is built in an analogous way and the outputs of both constructions are then joined in an OR gate that creates the final output of the miter. A complete example of a miter for the threshold value 00000101 is illustrated in Figure 5.1, the newly constructed part comparing the maximum error to the threshold consists of the logical gates connected to the outputs of the $G - C$ block. The following logical formulas describe the final miter structure:

$$w_1 = (e_1 \wedge e_2) \vee e_3 \vee e_4 \vee e_5 \vee e_6 \vee e_7$$

$$w_2 = ((\neg e_0 \vee \neg e_1) \wedge \neg e_2) \vee \neg e_3 \vee \neg e_4 \vee \neg e_5 \vee \neg e_6 \vee \neg e_7$$

$$WCRE(G, C) > 5 \iff (\neg sign \wedge w_1) \vee (sign \wedge \neg w_2)$$

The miter constructed by the proposed algorithm was compared to the classic miter featuring absolute value output and a comparator only detecting positive value. The experimental results of the area comparison is described in Section 6.2.1.

5.4 Satisfiability check procedure

The whole miter is built in the AIG representation and ABC's decision procedure `IvyProve` is used to determine, whether the miter's single output bit is satisfiable. This procedure is optimized to quickly solve the equivalence checking problem for combinational and sequential circuits.

Firstly, the miter circuit is converted into a formula in the Conjunctive Normal Form (CNF) and then the decision procedure is called. `IvyProve` is an iterative function and each iteration consists of several operations whose aim is to reduce the SAT problem complexity and to solve the instance in the shortest time possible. Each iteration has a certain resource limit and if the problem cannot be solved within this limit, the resource constraints for the next iteration is increased. The process is repeated until the problem is solved or a specified number of iterations reached. Every iteration consists of three basic operations:

- simulation,

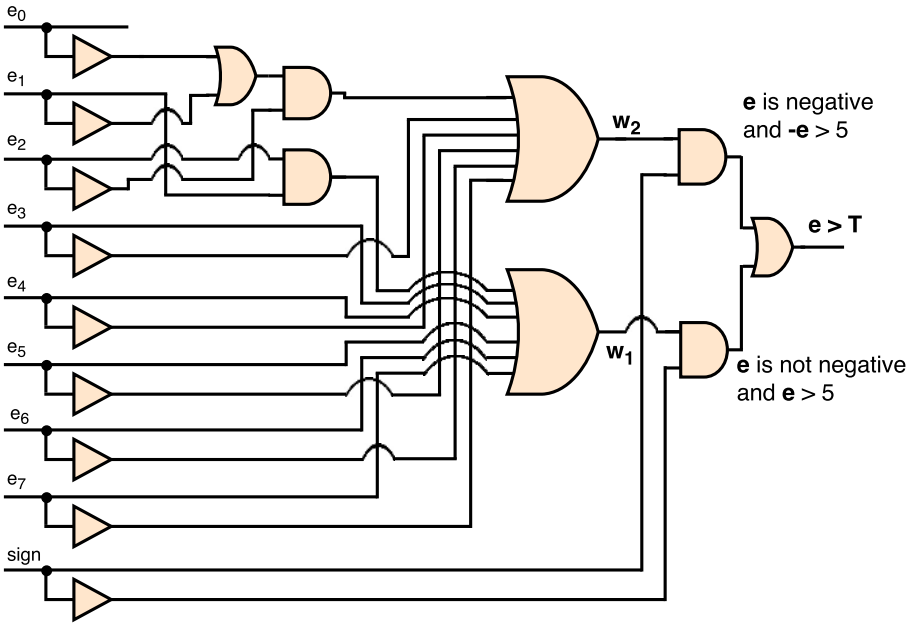

Figure 5.1: The miter with comparator constructed by the proposed algorithm for threshold $T = 5$.

- fraiging – solving intermediate node equivalence,
- mitering – solving output node equivalence / satisfiability.

The circuit **simulation** is employed to detect the nodes of the miter which are not functionally equivalent – nodes are not functionally equivalent, if their output differs for the same input vector. Simulation is executed until some groups (classes) of nodes appear to be equivalent. During **fraiging**, the candidates from a certain group are then checked by SAT solver and are a) merged is they are proved to be equivalent (miter is simplified) or b) proved to be different with a counterexample (input vector for which their outputs differ) returned from the solver. The counterexample and its modifications are then used as input vectors for the simulation with higher probability of distinguishing candidate equivalent nodes than random vectors. When the resource limit for simulation and fraiging is depleted, the algorithm moves on to the next stage – **mitering**.

The mitering is a classic satisfiability check of the circuit's output. In the CNF representation, every node of the miter is represented by a boolean variable. The decision procedure gradually tries to find a variable assignment that results in the satisfiablity of each clause and therefore the whole formula. When the procedure reaches a point where a variable would need to acquire both logical '1' and '0' in order to satisfy two different clauses, a conflict is registered and the procedure backtracks to a previous variable assignment. The defined resource limit for both fraiging and mitering procedure is the maximum number of conflicts on a single node of the miter. When the limit is reached, the procedure is terminated as unsuccessful (the problem could not be solved within the resource limit).

In case the problem was not solved with the current limit, fraiging with increased resources puts higher effort into miter simplification. In the fraiging round, the solver firstly tries to resolve potential equivalent classes that participated in most conflicts during the previously executed mitering. If the nodes are proved to be equivalent, the miter is

simplified by their merging and mitering is possibly accelerated because the problematic group of nodes was eliminated.

The iterative nature of the procedure with arbitrary resource limits allows us to optionally adapt the length of SAT queries and fulfill the goal proposed in Section 4.3. In the evolutionary cycle, only the candidates whose miter was proven unsatisfiable within the defined resource constraints are considered satisfactory solutions. If a miter of a solution is proven satisfiable, such solution features WCE exceeding threshold T and does not satisfy the error constraint. Finally, the solutions whose error could not be determined feature unfeasibly long WCE evaluation and their offspring would probably require the same or even higher evaluation time. In accordance with our verifiability-driven search strategy, such solutions are discarded immediately and their evolutionary branches are not explored. In Section 6.3 we show the enormous impact of the limited SAT calls on the success of the evolutionary algorithm.

5.5 Algorithm output and the best candidate solution

During the execution of the evolutionary algorithm, every improvement of the best candidate solution is recorded in the log file. Each record contains information summarized in the following list. An example shortened log file is included in the Appendix B. These records can be also generated periodically every X generations in case we need a periodic or a more detailed information about the course of the evolution. The period X is specified in the configuration file.

- Order number of the current generation,
- time when the generation was created,
- estimated area of the best candidate found so far.

After the predefined number of generations is executed or the time dedicated to the algorithm runs out, the best candidate solution found is written into output files in both verilog and chromosome representation. The chromosome can serve as the seed for a new evolution run in case we would need to start the algorithm from this particular approximate solution. Various run statistics are also included in the log file, the most interesting ones include:

- Size of the best solution and the estimated area savings.
- Total number of candidate evaluations.
- Average number of evaluations executed per second.
- Number of candidates whose evaluation could be skipped.
- Total counts of SAT, UNSAT and failed decision procedure calls.

These statistics are crucial to determine the right parameters for the CGP algorithm to achieve the best results possible and also serve to evaluate both the performance of the method and the impact of verifiability-driven scheme in comparison to the classical CGP search.

5.6 Pareto front approximation

Pareto optimality (or efficiency) is a state of allocation of resources from which it is impossible to reallocate in such a way that makes any one criterion better off without make at least other criterion worse off. In our case, there are two criteria for each candidate solution – $WCRE$ and size. A Pareto optimal solution for a given $WCRE$ is a circuit whose size is minimal and each reduction of the size would result in the increase of its error. The construction of the optimal Pareto fronts would require and exhaustive search of the whole state space of the problem, in other words evaluation of every candidate solution possible. Because such a task is obviously computationally unfeasible, we try to approximate the optimal Pareto front as closely as possible. There are two main approaches to the approximation – single and multi objective optimization.

In this work, we focus on the single-objective optimization, because it has already been shown that it generally produces better results than the multi-objective optimization [25]. The single metric of the circuit that serves as the fitness function is the circuit's size (others could include delay, energy consumption, etc.) and the objective is to minimize this metric. Other features that have to be enforced on the circuit are represented as constraints. Only a circuit that meets all the constraints is satisfactory and has the fitness value equal to its size. Other circuits have the fitness equal to $+\infty$ and thus are always worse than the feasible ones and never accepted.

The worst-case error constraint is set at the beginning of the evolution run and remains unchanged. The run then produces a single solution meeting the error limit (its $WCRE$ is usually very close to the limit). In order to construct a Pareto set of the approximate multipliers that represents the various trade-offs between the circuit's error and area, we need to perform multiple evolution runs with different $WCRE$ limits. Each such run produces a new solution with a certain $WCRE$ and a corresponding area savings. Eventually, the best solution for each $WCRE$ value is identified and together these chosen candidates make up the resulting Pareto set.

Chapter 6

Experiments and Results

6.1 Experimental setup

In order to evaluate the performance of the proposed method, we primarily focused on complex approximate multipliers as they are the most challenging benchmark problems. Because only 8-bit multipliers with guaranteed error bounds were presented in the literature so far, there are no solutions available for the direct comparison in the case of 16-bit and more complex approximate multipliers. We present Pareto sets (the error and key circuit parameters) for 20-bit, 24-bit, 28-bit, and 32-bit approximate multipliers and up to 128-bit approximate adders to demonstrate the scalability of the proposed method.

The array multipliers and ripple carry adders composed of 2-input gates were employed as the both the seeding and golden circuits for CGP. For each target $WCRE$, CGP was executed for 2 hours. Statistical evaluation is based on 30 independent runs. The accurate implementations were created by means of Verilog $*$ and $+$ operators and synthesized into netlist Verilog representation by freely available `yosys` synthesis tool. The size of the CGP gate grid was set to the number of gates of each evolved circuit and L-back parameter was set to the maximal value. The evolution was allowed to use the entire list of the logical gates mentioned in Section 5.2.

6.2 Performance comparison

Firstly, the performance of the implemented algorithm was compared to the freely available implementation of CGP computing fitness by the means of full simulation [23]. The algorithm employs parallel 64-bit simulation and compilation of the evaluated individual's chromosome into native assembly code. The comparison of the number of evaluations performed per second by this algorithm and the method proposed and implemented in this work is shown in Table 6.1. The experimental evaluation is conducted for multipliers of various bit widths with the maximum width of 16 bits for the proposed method and 12 bits for the full simulation. The tested CGP implementation employing simulation requires the circuit specification in form of a truth table. Such representation does not scale well for larger circuits – the truth table for 14-bit multiplier and above has the size of gigabytes and the manipulation with it becomes unfeasible.

Table 6.1 clearly shows that the fitness evaluation using simulation is much faster for smaller multiplier instances. However, with increasing input bit widths it runs into severe scalability problems. Fitness evaluation with SAT solver is inefficient for simple problem

	CGP native 64		Proposed algorithm	
Multiplier bit width	Evals / sec	Ratio	Evals / sec	Ratio
4	350360.24	—	413.49	—
6	34616.92	10.12	150.48	2.75
8	2162.04	16.01	77.64	1.95
10	108.45	19.94	55.19	1.40
12	4.99	21.73	22.03	2.51
14	—	—	17.17	1.28
16	—	—	10.36	1.66

Table 6.1: Comparison of the implemented algorithm and available implementation of CGP evaluating fitness by full simulation. Column Ratio shows the quotient of $(e*s_{i-1}^{-1})/(e*s_i^{-1})$.

instances because of the costly circuit transformation to CNF and solver utilization. In the Ratio columns, we can see that the proposed method scales better and maintains its performance for larger bit widths as well. The CGP implementation using simulation slows about 16 times for each two input bits added. On the other hand, the proposed method slows down only roughly two times. This allows us to successfully design much larger approximate multipliers. Note that with the utilization of the decision procedure limit to roughly one second described in the next Section, proposed algorithm always performs at least about one evaluation per second regardless of the circuit size.

The results given in Table 6.1 were computed as average values of 10 independent evolutionary runs for each multiplier bit width. All runs had the error threshold set to $WCRE = 10\%$ and were limited to 10 000 candidate evaluations.

6.2.1 Proposed miter area comparison

The miter designed by the proposed algorithm was compared the conventional miter with the absolute value computation. Table 6.2 shows the number of logical nodes and edges in both miters and the achieved node and edges savings. Both miters were built to compare two 64-bit vectors (e.g. outputs of two 32-bit multipliers). The miters were designed as Verilog netlists, loaded by ABC and optimized by the available ABC optimization procedures prior to recording statistics. We can see that the node savings are constant for all of the tested $WCRE$ thresholds. The edge savings differ slightly but are still significant. Overall, the smaller and simpler mitter representation results in a faster candidate evaluation and a higher CGP performance.

6.3 The impact of limited SAT resources

The first part of the experimental results illustrated in Figure 6.1 shows the importance of SAT solving resources limit on an example set of runs for 20-bit multiplier. In this experiment, we approximated the golden 20-bit multiplier for 9 target values of $WCRE$ from the set {0.1, 0.2, 0.5, 1, 2, 5, 10, 15 and 20%} and evaluated the performance of the proposed method with three different settings of the resource limit L controlling the maximal number of conflicts for one AIG node: (1) no limits, i.e., $L{=}\infty$, (2) $L{=}160k$, and

	Absolute value miter (A)		Proposed miter (B)		Savings B vs. A [%]	
$WCRE$ [%]	Nodes	Edges	Nodes	Edges	Nodes	Edges
$1 * 10^{-4}$	1383	2588	758	1643	45.2	36.5
$1 * 10^{-3}$	1383	2675	758	1817	45.2	32.1
$1 * 10^{-2}$	1383	2738	758	1944	45.2	29.0
0.1	1383	2694	758	1741	45.2	35.4
0.5	1383	2821	758	1992	45.2	29.4

Table 6.2: Comparison of the size of classic miter with absolute value and the newly proposed miter.

(3) L=20k. The meaning and usage of these limits was described in Section 5.4. Note that the limits $L = 160k$ and $L = 20k$ roughly correspond to the time limit of 20 sec. and 1 sec., respectively, on 20-bit multipliers.

We can see that the results of different limits are almost identical for larger worst case relative error (WCRE). However, in case of smaller errors, the unlimited SAT resources runs and the $L = 160k$ runs (blue and purple in Figure 6.1) have not achieved almost any distinct improvements. This was caused by the fact that the evolution explored candidate solutions that featured unfeasibly long fitness evaluation (miter satisfiability solving). Such candidates are very likely to produce offspring with the same or even longer evaluation. The evolutionary runs exploring these branches of candidate solutions slow down drastically and perform only small amount of candidate evaluations during the 2 hour limit. The long evaluation process is caused by changes of the candidates whose structure is gradually more distinct from the golden solution. The fraiging stage of the decision procedure fails to simplify the miter which then takes very long time to examine.

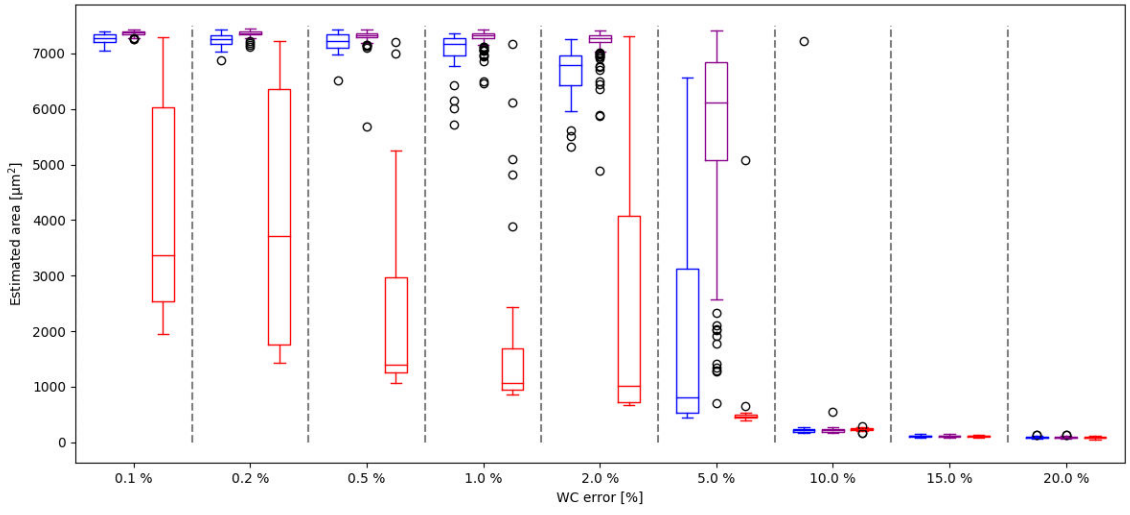

Figure 6.1: The comparison of the final circuit's area of unlimited (blue), 160k limited (purple) and 20k limited (red) SAT evaluation evolutionary runs. The plots are sorted in the same respective order for each target $WCRE$.

Solving this problem is the key point where our verifiability-driven approach brings enormous improvement of the search method. We set the resource limit R of the ABC's decision procedure to $L = 20k$ maximum conflicts at a single node of the miter. The comparison of executed evaluations per second of different SAT solver limits for the approximation of 20-bit multiplier is summarized in Table 6.3. We can see that setting the maximum conflicts limit to 160k (roughly 20 seconds evaluation maximum) did not bring almost any significant improvement. On the other hand, the 20k limit performed much better mainly for runs with smaller error threshold. All runs show similar performance for $WCRE \geq 10\%$.

The $L = 20k$ limited SAT runs (red in Figure 6.1) avoided unverifiable solutions and constructed a series of promptly evaluable candidate solutions leading to the near optimum final solutions. Because of the huge performance improvement, this became one of the key features of the designed method and all other experiments mentioned in this section are performed with SAT resource caps that limit the SAT solving procedure to roughly one second at maximum (20k limit).

To better show the differences between the runs, we include brief statistics of the SAT solving procedure calls and its results. For example, if WCRE = 0.1% and $L = 20k$, 22,050 SAT calls were produced and 11% of them were terminated on average because of the termination condition. In the case of $L = 160k$, 856 SAT calls were produced only (15% terminated). The average number of SAT calls (across all target errors) that were forced to terminate is 6.28% (for $L = 160k$) and 8.84% (for $L = 20k$). If $L{=}\infty$, 170 SAT calls were evaluated for WCRE = 0.1% only. Despite the fact that some potentially good candidate circuits are quickly rejected, the aggressive resource limits allowed us to generate and evaluate significantly more candidate circuits and thus to substantially improve the quality of the results.

WCRE [%]	**Unlimited**	**160k limit**	**20k limit**
0.1	0.084	0.110	1.457
0.2	0.074	0.088	2.048
0.5	0.079	0.097	4.55
1.0	0.078	0.098	6.227
2.0	0.098	0.113	5.304
5.0	8.083	0.589	35.807
10.0	771.983	760.411	621.132
15.0	1827.67	1253.813	1107.597
20.0	2209.630	2254.037	1851.714

Table 6.3: Comparison of the average executed evaluations per second of unlimited SAT calls and SAT calls limited to maximum 160k and 20k conflicts used to approximate 20-bit multipliers.

The values given in Table 6.3 were obtained from 30 independent evolutionary runs for each parameter settings. Each run was limited to either 1 million generations or 2 hour computational time (first condition met ends the run). Note that higher error threshold results in more evaluations per seconds measured, because the candidate solutions are

reduced quickly and their smaller size leads to the faster miter evaluation and the higher amount of neutral mutations.

We can see that the best solutions found by the limited SAT call runs shown red in Figure 6.1 form a smooth curve with a stable trend. From this fact we can assume that even though some of the correct candidate solutions were left out, the evolution was still able to find a sequence of quickly verifiable solutions that leads to a near optimum final solution. Given enough computational time, the unlimited runs would most likely produce very similar results (respecting the curve of the best solutions) without any significant improvement. This observation is very interesting from the evolutionary point of view, when the CGP search method can achieve similar results with restricted search space.

6.4 Approximate multipliers

The main aim of this study is to show that the proposed method is scalable and can approximate complex multipliers. We present the results of the approximation process on 12-bit, 16-bit, 20-bit, 24-bit, 28-bit and 32-bit multipliers. The target $WCRE$s were adopted accordingly to respect the range of values in a given bit width. We used the same setup as in the previous sections, but had to increase the time of optimization to 4 hours for the 24-bit multiplier and 6 hours for larger multipliers. The reason is that the search space becomes much bigger. Note that while the exact 12-bit multiplier contains 850 two-input gates, the 32-bit exact multiplier requires over 6,300 gates. We obtained (as the result of evolution) and analyzed over 1190 unique multipliers which are summarized into box plots and Pareto fronts.

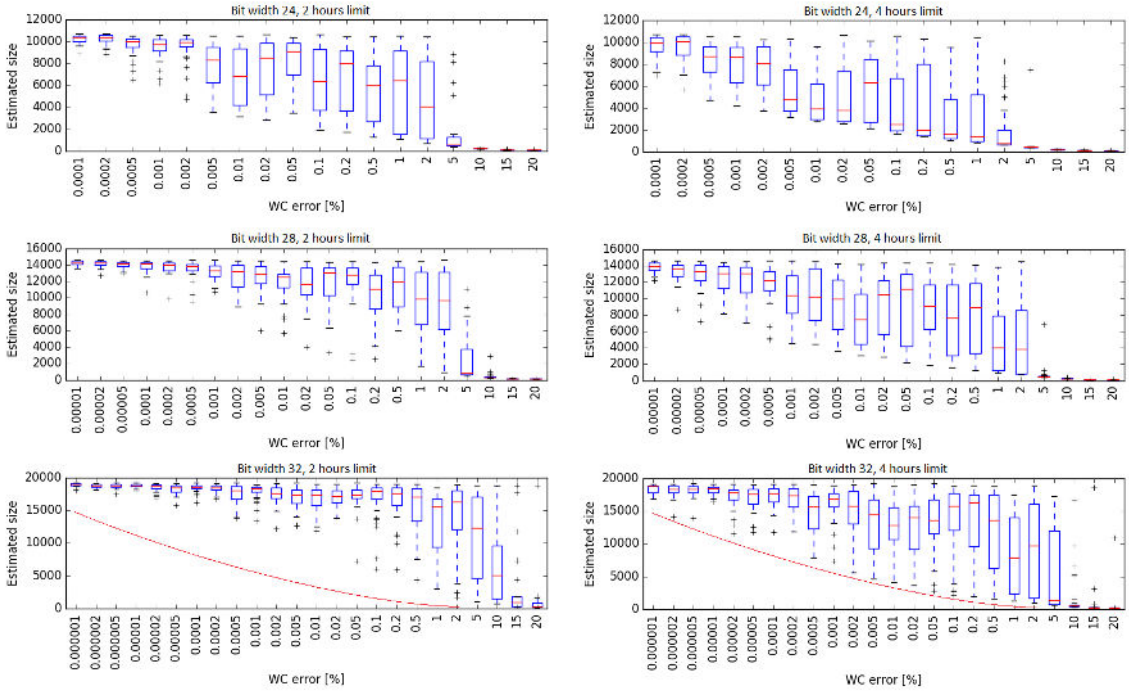

Figure 6.2: Comparison of evolved 24, 28 and 32-bit multipliers from 2 and 4 hours limited evolutionary runs. Red curve marks imaginary Pareto front for 32-bit multipliers.

Figure 6.2 shows the search quality of evolutionary runs limited to 2 and 4 hours for complex multipliers (24, 28 and 32-bit). The imaginary optimal Pareto front for 32-bit multipliers is represented by the red curve in the corresponding plots. The graph clearly illustrates the fact, that 2 hour limit runs successfully approximate Pareto fronts only for $WCRE \geq 5$. However, these runs are too short to find such solutions for smaller $WCRE$ values. The multipliers evolved by 4 hours limited experiments achieve much better Pareto front approximation, but it is clear that the search is still not complete. Therefore we employed 6 hours evolutionary runs for 28 and 32-bit multipliers to acquire the best results possible.

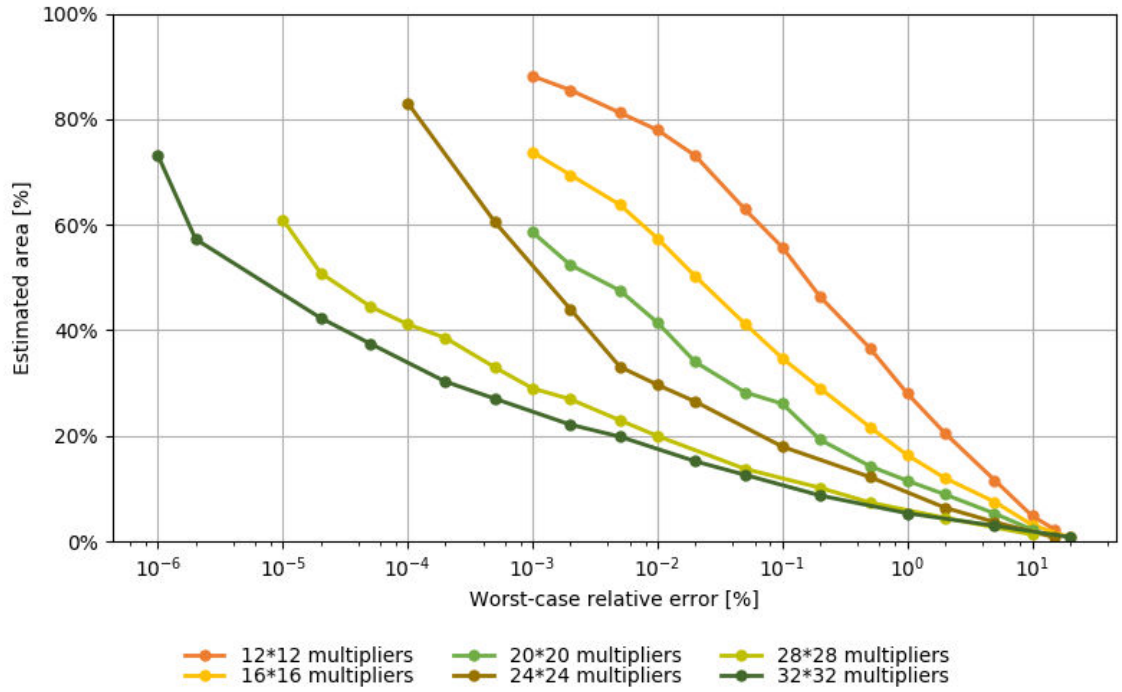

Figure 6.3: The estimated area savings for Pareto fronts of various bit width approximate multipliers. 100 % refers to the corresponding area of the accurate multipliers for a given bit width.

The resultant Pareto fronts of the various bit width multipliers are shown in Figure 6.3. Each of the curves in the graph shows an identical trend – the bigger and more complex the circuit is, the more relative area can be saved for the same target $WCRE$. The comparison of Figure 6.3 providing the Pareto fronts for the area estimation to Figure 6.7 illustrating the circuit area after the synthesis clearly shows the connection of the two metrics. The plots for corresponding multiplier bit widths demonstrate almost the same course which means that the employed method of circuit size approximation is suitable for our purposes.

To determine the limits of the proposed algorithm, we tried to create the approximations of 48-bit multiplier. This experiment proved unsuccessful – the evolutionary runs with 6 hour time limit did not achieve any significant improvements for neither of the examined error values. The feasible design of 48-bit multipliers would likely require different CGP parameterization and much longer available computational time. Thus we conclude the multiplier approximation experiments with the claim that the proposed algorithm can be successfully employed in the design of up to 32-bit approximate multipliers.

6.5 Approximate adders

In order to demonstrate that the proposed method is applicable and scalable for other complex arithmetic circuits as well, we constructed Pareto sets for various approximate adders with 20 to 128 bit operands. Approximation of adders is much easier than approximation of multipliers, since adders are structurally less complicated and the number of outputs is smaller. For example, the exact 20-bit adder requires 140 two-input gates and the 128-bit adder consists of 1,000 gates (more than six times less than the exact 32-bit multiplier). The approximate adders were constructed using the the same setup as in the previous section. A single CGP run took 2 hours (for all bit widths). Figure 6.4 shows the estimated area of approximate adders occupying corresponding Pareto sets. The lines corresponding to 64 and 128 adders are not as steep as the lines for smaller adders because of the structural features of the adders. The long carry chains of 128-bit multiplier do not allow evolution to save significant amount of circuit size for minor error thresholds.

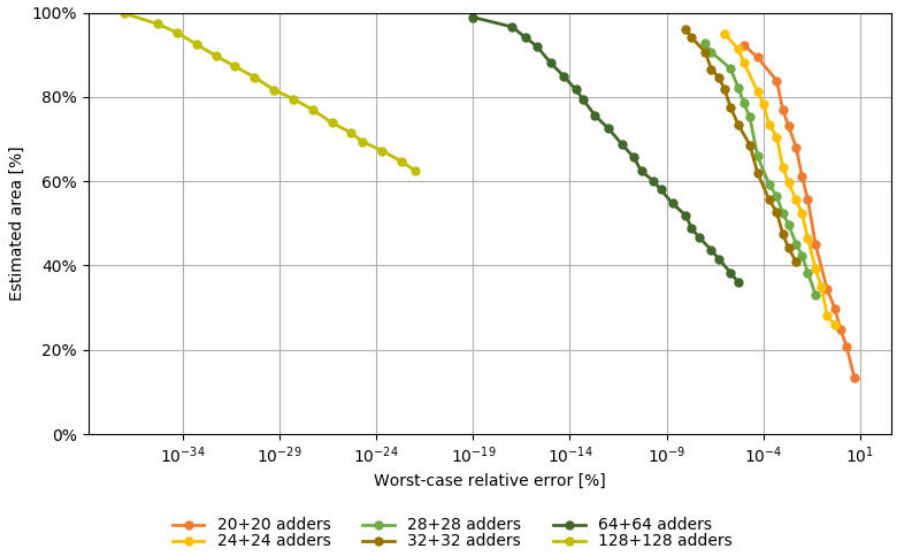

Figure 6.4: The estimated area for the Pareto fronts of various bit width approximate adders.

Similarly to the previous section, Figure 6.4 can be compared to the results of the circuit synthesis in Figure 6.8. Note that the Pareto fronts of the synthesized circuits contain less individuals than the Pareto fronts of circuits with estimated area. This is caused by the optimizations performed by the synthesis tool which uses bigger functional blocks (e.g. full adder) to achieve better results while the area estimation only works on the logical gate level.

6.6 Circuit synthesis results

The results published in this section were achieved with the significant help of the research group of Lukas Sekanina and Zdenek Vasicek, namely Vojtech Mrazek provided much appreciated help, support and knowledge during the testing and synthesis of circuits into the target technology. The results are included in order to show comparison to other approxi-

mate solutions found by different algorithms and demonstrate the similarity between area estimations computed in the CGP algorithm and the area of the synthesized solutions.

The 5 best-scored circuits for each $WCRE$ were at the end of evolution synthetized using Synopsys Design Compiler (high-effort compiling for better quality of results) for a 45 nm technological library in order to obtain non-functional parameters such as the area and power-delay product (PDP). A combinational circuit can usually be synthesized to either have minimal power consumption or minimal delay (length of the longest combinational path in the circuit). Since there is a trade-off between the two metrics, power consumption and delay of the circuit are often joined into one metric – power-delay product – to better show the circuit's power efficiency.

6.6.1 SAT resource limit impact

We will now demonstrate that the SAT resource limits are crucial for the results of final circuit synthesis as well. The first synthesis experiment shows the parameters of the best evolved 16-bit multipliers for the three examined SAT procedure resource limits – unlimited, $L = 160k$ and $L = 20k$ maximum conflicts. Figure 6.5 demonstrates evolved circuit parameters very similar to Figure 6.1, which is plotted for 20-bit multiplier. All limits show almost identical performance for higher error thresholds. However, the difference between the evolved solutions grows for smaller $WCRE$. The unlimited and $L = 160k$ limited runs did not achieve any improvements for $WCRE = 0.1\%$ while the $L = 20k$ runs keep approximating the optimal Pareto fronts well.

Note that the parameters of some approximate multipliers shown in Figure 6.5 are worse than for the accurate multiplier. The reason is that the relative area is the only non-functional circuit parameter optimized by CGP while the PDP and area are computed at the end of the optimization using the Synopsys Design Compiler. We have never observed this discrepancy for the limit L=20K.

6.6.2 16-bit multiplier comparison

In this section, the 16-bit approximate multipliers that were generated by the means of the proposed method are compared to approximate multipliers available in the literature. Recall that the approximation error of these multipliers was estimated using the random simulation. In order to perform a fair comparison, we determined $WCRE$ for the multipliers from the literature using iterative calls of the proposed method with different target $WCRE$ (from 10^{-3} up to 20%). We also provide MAE parameter, note that it is obtained using the random simulation (10^9 input vectors) without formal guarantees.

The following 16-bit approximate multipliers were considered in this study:

M1 Approximate configurable multipliers from the lpACLib library [20]. In this case, the multiplication is recursively simplified using two different variants (denoted as *Lit* and *V1*) of an elementary block representing a 2-bit multiplier. The partial results are summed using accurate adders. We implemented 32 different architectures consisting of four 8-bit multipliers where each of these multipliers is configurable as exact/approximate (2^4 configurations) and can be built using either *Lit* (M1Lit) or *V1* (M1V1) blocks.

M2 The approximate multiplier employing the bit-significance-driven logic compression as introduced in [18].

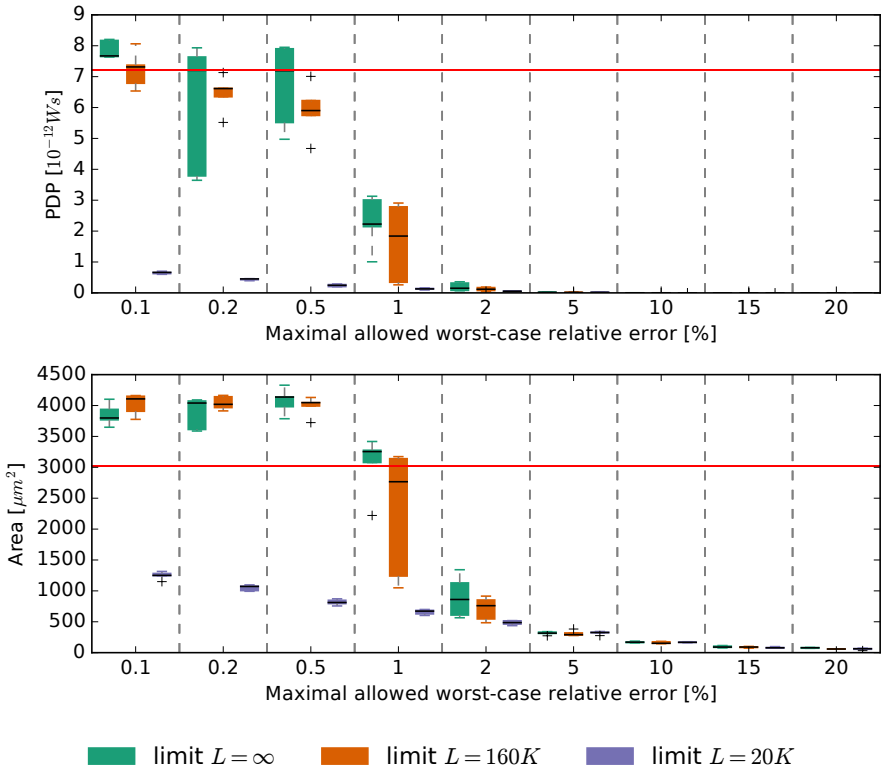

Figure 6.5: PDP and area of approximate 16-bit multipliers for 9 target errors obtained using 3 different resource limits L on the SAT solver. The red line shows the PDP and area of the accurate multiplier.

M3 Approximate multipliers obtained from exact multipliers using the bit-width reduction. The reduction replaces 16-bit multipliers by accurate 15-bit, 14-bit, etc. multipliers and leaves the LSBs of the operands zero.

M4 The approximate multiplier composed of approximate 2-bit multipliers as proposed in [9].

M5 Approximate multipliers composed of 8-bit multipliers that are available in the EvoApproxLib library [16]. The construction principle is taken from [9].

For all considered multipliers, the value of power-delay product (PDP) is plotted against $WCRE$ and MAE in Figure 6.6 (only Pareto sets are visualized). While the simple truncation provides the same quality of results as the proposed method for large target errors (up to 10% $WCRE$), it is significantly outperformed by our approach for small target errors. Despite that the existing approximate multipliers typically exhibit good trade-offs between the error and PDP in specific applications (as demonstrated in the relevant literature), Figure 6.6 clearly shows that these multipliers are considerably Pareto-dominated by the multipliers obtained using our approach.

6.6.3 Complex approximate multipliers synthesis

For each target $WCRE$ and each multiplier bit width, the five best evolved individuals were synthesized into the target 45 nm technology in order to obtain their real on-chip

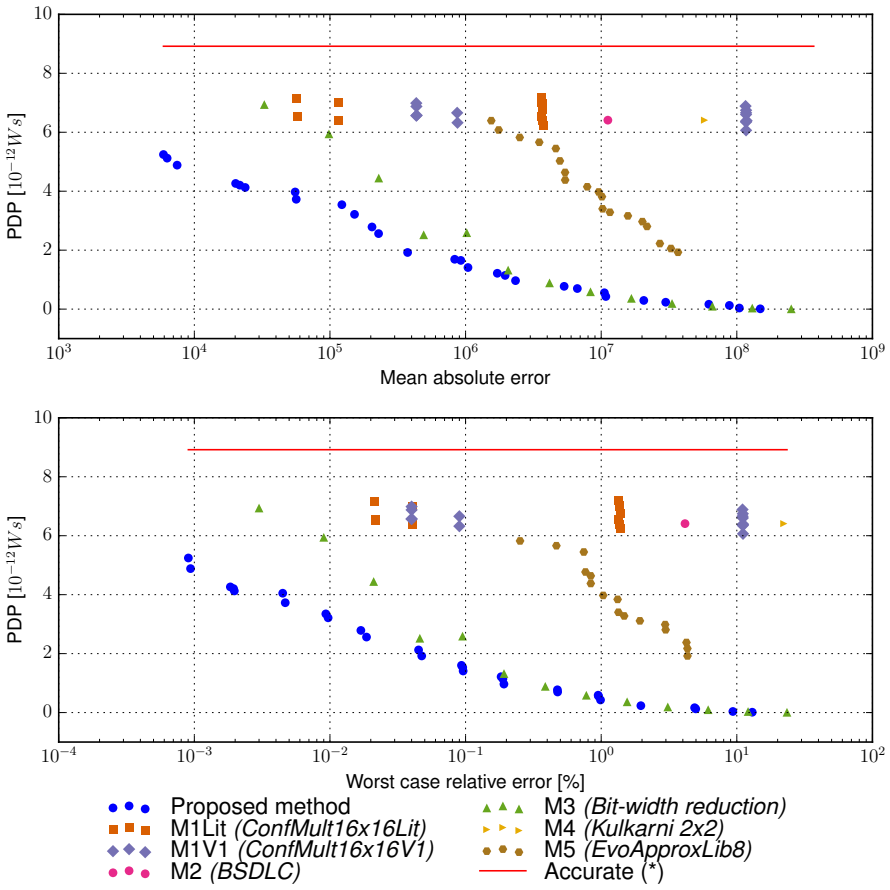

Figure 6.6: Comparison of available 16-bit approximate multipliers to the results of the proposed method for both area and PDP.

parameters – area, power consumption and delay. These parameters are plotted against $WCRE$ in Figure 6.7. Another error metric – mean absolute error (MAE) – is included as well to show the correspondence between the circuit's worst case error and its mean error. The curves for both $WCRE$ and MAE have similar course, however, the fronts for MAE are separated because this metric shows the absolute error. Obviously, the absolute error of 32-bit multiplier is much higher than the absolute error of 12-bit multiplier for the same target $WCRE$.

The leftmost plot in Figure 6.7 showing the dependency of the real circuit area and $WCRE$ can be compared to Figure 6.3. The correspondence of the two figures clearly shows the suitability of the employed candidate area estimation.

6.6.4 Complex approximate adders synthesis

For the experiment completeness, we also include the synthesis results of designed approximate adders in Figure 6.8. Once again, we can see that the Pareto fronts for 64 and 128-bit adders do not feature as steep area and PDP improvements as the fronts for smaller bit widths. This is caused by the structure of the adders where the long carry chains in the adders make the evolution less effective during the area reduction. Also note that the

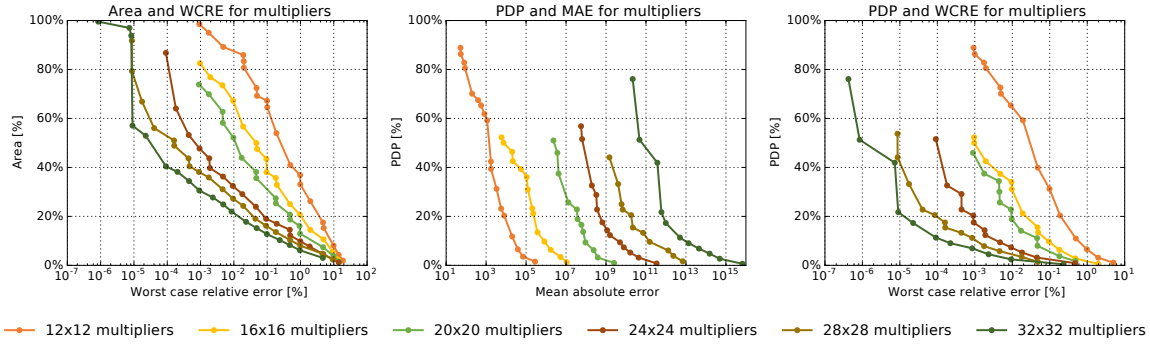

Figure 6.7: Pareto sets showing the area and PDP parameters of evolved approximate multipliers. 100 % refers to the corresponding parameters of the accurate multipliers for a given bit width.

Pareto fronts after the synthesis contain less individuals than the fronts constructed from adders with estimated area (Figure 6.4). Some of the not dominated solutions from the first front are dominated by other candidates with greater estimated but lesser real on-chip area. We can assume that some of the evolved solutions have suitable structural qualities for the synthesis tool and its optimization procedures while others cannot be optimized as much. However, to assess these qualities during the evolution would require the employment of the synthesis tool in each evaluation of the candidate solutions which is computationally unfeasible.

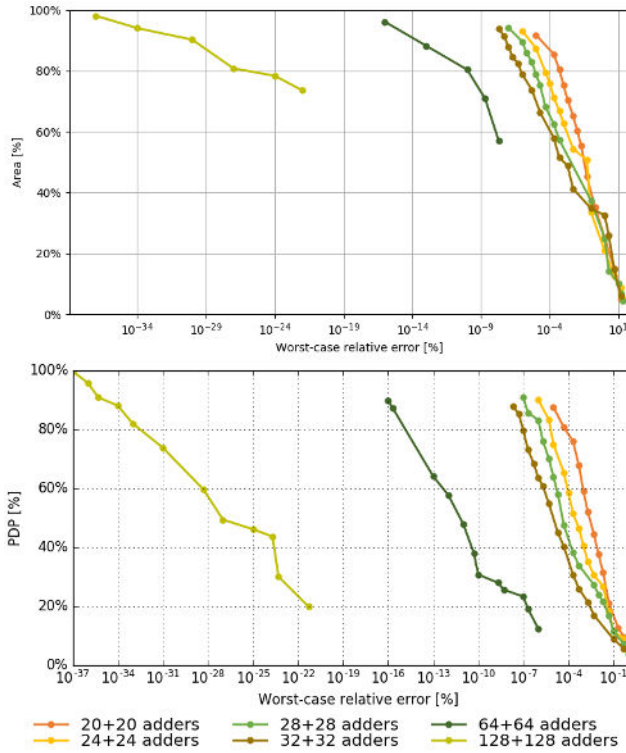

Figure 6.8: Pareto sets showing the size and PDP of evolved approximate adders after synthesis. 100% refers to parameters of the accurate adder for a given bit width.

Chapter 7

Conclusion

Approximate computing is a quickly developing methodology and is currently subject of intensive research. The main aim of this thesis was to examine the possibilities of the utilization of formal verification methods for relaxed equivalence checking in the approximate combinational circuits design. To fulfill this goal, we propose a new miter construction that allows for efficient approximate equivalence checking tailored to search-based approximation of complex arithmetic circuits.

Moreover, we design a novel search strategy for the synthesis of approximate circuits with formal error guarantees that integrates the Cartesian Genetic Programming and the approximate equivalence checking. Using a resource-limited verifier, the strategy drives the search towards promptly verifiable candidates and thus provides scalable approximation of complex circuits.

The proposed search strategy and miter construction were implemented as a module of the ABC synthesis and verification tool. With this implementation we performed an extensive experimental evaluation of our approach on complex circuit instances – 12 to 32-bit approximate multipliers and 20 to 128-bit approximate adders. The experimental results demonstrate the scalability and efficiency of the miter and the search strategy.

The existing methods for the combinational circuit approximation struggle to design larger than 8-bit approximate multipliers with formal error guarantees and 16-bit multipliers without formal guarantees. As shown in Chapter 6, we were able to construct Pareto sets of 128-bit adders and 32-bit multipliers that represent the trade-offs between the circuit error and size. All of the candidates in the Pareto sets are developed with formal guarantees on the approximation error. Such performance is unpreceded and this is the first time such Pareto sets have been constructed. The developed approximate circuits can also serve as model benchmark problems for the performance evaluation of new algorithms. Our method thus significantly improves the capabilities of the existing methods.

The results of this thesis were summarized in a paper, presented at the Excel@FIT 2017 conference and were awarded a prize for the design and implementation of novel technique which significantly enhances the performance of approximate circuit design methods. An improved version of the paper was also submitted to the International Conference on Computer Aided Design ICCAD 2017.

The future work on this approach will include an extensive examination of the impact of limited SAT calls on the success of the search strategy. The limits studied in this thesis have rather experimental character and further research can give more insight into their effects and further improve the performance. We will also explore how to utilize the constructed circuits in the efficiency and energy-aware applications.

Bibliography

[1] Barrett, C.; Sebastiani, R.; Seshia, S.; et al.: *Satisfiability Modulo Theories*. IOS Press. 2009. ISBN 978-1-58603-929-5. pp. 825-885.

[2] Bäck, T.: *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press. 1996.

[3] Bryant, R. E.: *Graph-Based Algorithms for Boolean Function Manipulation*. In IEEE Trans. Comput., vol. 35. IEEE Computer Society. 1986. pp. 677-691.

[4] Burch, J. R.; Clarke, E. M.; McMillan, K. L.; et al.: *Symbolic model checking*. In Information and Computation, vol. 98. IEEE Computer Society. 1992. pp. 142-170.

[5] Chandrasekharan, A.; Soeken, M.; Grosse, D.; et al.: *Precise error determination of approximated components in sequential circuits with model checking*. In Proc. of DAC'16. ACM. 2016.

[6] Clarke, E. M.; Emerson, E. A.: *Design and synthesis of synchronization skeletons using branching time temporal logic*. In Proc. of the IBM Workshop on Logics of Programs, vol. 131. Springer-Verlag. 1991. pp. 52-71.

[7] Cook, S. A.: *The Complexity of Theorem-proving Procedures*. ACM. 1971. pp. 151-158.

[8] Jouppi, N.: *Google supercharges machine learning tasks with TPU custom chip*. [Online; visited 27.12.2016].
Retrieved from: https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html

[9] Kulkarni, P.; Gupta, P.; Ercegovac, M. D.: *Trading Accuracy for Power in a Multiplier Architecture*. In J. Low Power Electronics, vol. 7. IEEE. 2011. pp. 490-501.

[10] Kulkarni, P.; Gupta, P.; Ercegovac, M. D.: *Trading accuracy for power with an underdesigned multiplier architecture*. VLSI Design. 2011. pp. 346–351.

[11] Miller, J.; Thomson, P.: *Cartesian Genetic Programming*. Springer. 2000. ISBN 80-85615-77-0.

[12] Miller, J. F.; Job, D.; Vassilev, V.: *Towards the Automatic Design of More Efficient Digital Circuits*. In Proc. of the 2nd NASA/DOD Workshop on Evolvable Hardware. IEEE Press. 2000. pp. 151-160.

[13] Mischenko, A.: *ABC: A System for Sequential Synthesis and Verification*. [Online; visited 20.12.2016].
Retrieved from: https://people.eecs.berkeley.edu/~alanmi/abc/

[14] Mittal, S.: *A Survey of Techniques for Approximate Computing*. In ACM Comput. Surv., vol. 48. ACM. 2016.

[15] Motwani, R.; Raghavan, P.: *Randomized Algorithms*. Cambridge University Press. 1995. ISBN 0-521-47465-5.

[16] Mrazek, V.; Hrbacek, R.; et al.: *EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods*. In Proc. of DATE'17. IEEE. 2017. pp. 258-261.

[17] Nelson, C. G.; Oppen, D. C.: *Simplification by cooperating decision procedures*. TOPLAS. 1979. pp. 245-257.

[18] Qiqieh, I.; Shafik, R.; et al.: *Energy-Efficient Approximate Multiplier Design using Bit Significance-Driven Logic Compression*. In Proc. of DATE'17. IEEE. 2017.

[19] Sekanina, L.: *Introduction to approximate computing: Embedded tutorial*. In Design and Diagnostics of Electronic Circuits And Systems (DDECS). IEEE. 2016. ISBN 978-1-5090-2467-4.

[20] Shafique, M.; Ahmad, W.; et al.: *A Low Latency Generic Accuracy Configurable Adder*. In Proc. of DAC'15. ACM. 2015. 86:1-86:6 pp.

[21] Tseitin, G. S.: *On the complexity of derivations in the propositional calculus*. In Studies in Mathematics and Mathematical Logic. Nauka. 1968. pp. 115-125.

[22] Vašíček, Z.; Sekanina, L.: *Evoluční návrh kombinačních obvodů*. In Elektrorevue vol. 43. Elektrorevue. 2004.

[23] Vasicek, Z.: *Efficient Phenotype Evaluation in Cartesian Genetic Programming*. [Online; visited 24.3.2017].
Retrieved from: http://www.fit.vutbr.cz/~vasicek/cgp/

[24] Vasicek, Z.; Mrazek, V.; Sekanina, L.: *Towards Low Power Approximate DCT Architecture for HEVC standard*. In Proc. of DATE'17. IEEE. 2017. pp. 1576–1581.

[25] Vasicek, Z.; Sekanina, L.: *Circuit Approximation Using Single- and Multi-objective Cartesian GP*. In Proc. of the 18th European Conference on Genetic Programming. Springer. 2015. ISBN 978-3-319-16501-1.

[26] Venkataramani, S.; Roy, K.; Raghunathan, A.: *Substitute and simplify: a unified design paradigm for approximate and quality configurable circuits*. In Proc. of DATE'12. IEEE. 2013.

[27] Venkataramani, S.; Sabne, A.; Kozhikkottu, V.; et al.: *SALSA: Systematic Logic Synthesis of Approximate Circuits*. In Proc. of DAC'12. IEEE. 2012.

Appendix A

Example configuration file

```
GENERATIONS  10000  # number of generations in each CGP run
RUNS         10     # number of CGP runs executed
MAX_ERR_PERC 10     # max percentual error of a candidate solution

PARAM_M 600         # number of collumns
PARAM_N 1           # number of rows
L_BACK  600         # level back connectivity

PARAM_IN  20        # number of primary inputs
PARAM_OUT 20        # number of primary outputs

POP_MAX      2      # maximal size of population
MUTATION_MAX 12     # maximum number of geners altered in one generation
FUNCTIONS    9      # 1-9 functions used to create the candidate solution

MODULE_NAME multABC
WRITE_LOG   1
PARAM_LOG   20000   # periodic log each X generations

LOG_F         ../log/perf.log   # output file for performance results information
CIRC_F        ../log/circ       # output file for the final circuit representation

TECHLIB_F ../synthesis/gscl45nm.lib# liberty file

MAX_RUN_TIME 7200   # max. time for each evolutionary run
MAX_ALG_TIME 10000  # max. time for the whole algorithm run

# file with verilog netlist representation of golden solution
GOLDEN_F     ../synthesis/mult10/mult10_synth_rmc.v

# file with verilog netlist representation of a subtractor with adequate width
SUBTRACTOR_F ../synthesis/sub20/sub20_synth_rmc.v

# first population seeded with given solution or random solutions
SEEDED               1

# file with the chromosome representation of the seeding circuit
SEED_F        ../synthesis/mult10/mult10.chr
```

Appendix B

Example log file

```
=========================================================
= START OF EVOLUTION CYCLE 1 OF 10, GENERATIONS: 10000
=========================================================

GENERATION: 0 (T: 2.3) -- fit = 1, area = 1053.4
GENERATION: 4 (T: 2.4) -- fit = 6553, area = 1051.1
GENERATION: 7 (T: 2.5) -- fit = 6553, area = 1048.8
GENERATION: 8 (T: 2.5) -- fit = 6553, area = 1046.5
GENERATION: 18 (T: 2.6) -- fit = 6553, area = 1041.9
GENERATION: 33 (T: 2.9) -- fit = 6553, area = 1030.4
GENERATION: 35 (T: 3.0) -- fit = 6553, area = 1026.7
GENERATION: 45 (T: 3.2) -- fit = 6553, area = 1024.4
GENERATION: 52 (T: 3.3) -- fit = 6553, area = 1023.9
GENERATION: 55 (T: 3.3) -- fit = 6553, area = 1021.6
GENERATION: 71 (T: 3.6) -- fit = 6553, area = 1019.3
GENERATION: 75 (T: 3.7) -- fit = 6553, area = 1006.9
GENERATION: 108 (T: 4.3) -- fit = 6553, area = 1000.0
GENERATION: 128 (T: 4.6) -- fit = 6553, area = 995.4
GENERATION: 133 (T: 4.7) -- fit = 6553, area = 993.1
GENERATION: 143 (T: 5.1) -- fit = 6553, area = 990.8
GENERATION: 149 (T: 5.1) -- fit = 6553, area = 981.6

=========================================================
= END OF EVOLUTION CYCLE 1 OF 10
=========================================================

Bit width:     8
Original area: 1053
Max error:     10.0000000000 %
Runs:          10
Generations:   10000
Mutations:     8

Evals performed:        637
SAT instances cnt:      401
UNSAT instances cnt:    134
Skipped instances cnt:  0
Total SAT invocatinos:  535

Time measured = 13.26 sec
Evals / sec   = 47.10
```