OptCE: A Counterexample-Guided Inductive Optimization Solver

Higo F. Albuquerque[1], Rodrigo F. Araújo[2], Iury V. Bessa[1], Lucas C. Cordeiro[1,3] and Eddie B. de Lima Filho[1,4]

[1]Federal University of Amazonas, Manaus, Brazil
[2]Federal Institute of Amazonas, Manaus, Brazil
[3]University of Oxford, Oxford, United Kingdom
[4]Samsung Electronics

Abstract. This paper presents optimization through counterexamples (OptCE), which is a verification tool developed for optimizing target functions. In particular, OptCE employs bounded model checking techniques based on boolean satisfiability and satisfiability modulo theories, which are able to obtain global minima of convex and non-convex functions. OptCE is implemented in C/C++, performs all optimization steps automatically, and iteratively analyzes counterexamples, in order to inductively achieve global optimization based on a verification oracle. Experimental results show that OptCE can effectively find optimal solutions for all evaluated benchmarks, while traditional techniques are usually trapped by local minima.

1 Introduction

Optimization is a tool employed in several research fields, such as biology (*e.g.*, biomolecular modeling energy functions) [1], computer science (*e.g.*, complexity reduction) [2], engineering (*e.g.*, filter design for digital signal processing) [3], and business (*e.g.*, profit increase) [4], with the goal of obtaining maximum system performance. Although there are several available optimization techniques (*e.g.*, simulated annealing [5], particle swarm [6], and genetic algorithms [7]), their main difficulty lies on locating the global minima of functions. As a consequence, they often present suboptimal solutions, *i.e.*, they are trapped by local minima, which commonly lead to low performance [8].

The present work introduces a tool based on the counterexample-guided inductive optimization (CEGIO) algorithms proposed by Araújo *et al.* [9,10], which is named as Optimization through Counter-Examples (OptCE). Indeed, OptCE is a tool instantiation of the approach developed by Araújo *et al.* [9], which now presents further evaluation regarding other function classes (broader applicability) and verifiers. OptCE is inspired by syntax-guided synthesis (SyGuS) and performs *inductive generalization* based on counterexamples provided by a verification oracle [11]. In particular, OptCE employs non-deterministic representation of decision variables and then iteratively constrains the state-space search based on counterexamples produced by boolean satisfiability (SAT) or

satisfiability modulo theories (SMT) solvers via inductive generalization, *i.e.*, OptCE exploits the counterexample provided by the solver to achieve complete global optimization [3] about an objective function.

The mentioned techniques (CEGIO) do employ model checking based verification procedures to guide the global convergence and extract information from counterexamples. Unlike meta-heuristic optimization techniques (*e.g.*, genetic algorithms and simulated annealing), CEGIO always finds the global minima for all evaluated benchmarks, which is also true for the benchmarks evaluated by Araújo *et al.* [9,10]. In addition, OptCE requires only one file with the specification and constraints for a given objective function.

Although the resulting optimization times associated to the approach employed here are often higher than what is obtained with other traditional techniques [10], the present inductive optimization technique based on the counterexamples guarantees global coverage and is capable of handling convex and non-convex functions, since it performs inductive generalization based on the counterexamples provided by a verification oracle [12]. Our main novel contributions are:

- Development of the first CEGIO-based tool that is able to perform global optimization of several function classes (*e.g.*, convex, discontinuous, nonlinear, and non-convex);
- Extensive experimental evaluation of the CEGIO algorithms;
- Comparison regarding optimization performances provided by different verifiers (CBMC [13] and ESBMC [14]) and SAT/SMT solvers (MathSAT [15], Z3 [16], Boolector [17], and MiniSAT [18]).

Our experiments are based on a set of publicly available benchmarks and all related tools, scripts, benchmarks, and results can be obtained online through this link `http://esbmc.org/benchmarks/optce.zip`.

2 Inductive Optimization Based on Counterexamples

OptCE is an optimization tool based on CEGIO, which processes a function through three basic steps: modeling, specification, and verification. In order to illustrate the OptCE's optimization process, we consider the *adjiman* test objective function in Eq. (1) and its minimization process.

$$f(x_1, x_2) = cos(x_1)sin(x_2) - \frac{x_1}{x_2^2 + 1}. \tag{1}$$

In particular, the *adjiman* function is a non-convex, non-separable, and differentiable function; it is defined on 2-dimensional space [19].

(i) **Modeling.** In the modeling step, the optimization problem is defined for a cost function (*e.g.*, Eq. (1)) and then its constraints are introduced, in order to avoid the state-space explosion in model checking [20]. Regarding the

Eq. (1), the optimization problem with its associated restrictions is described in Eq. (2).

$$\begin{array}{ll} \min & f(x_1, x_2) \\ \text{s.t.} & -1 \leq x_1 \leq 2 \\ & -1 \leq x_2 \leq 1. \end{array} \tag{2}$$

In particular, the optimization problems are modeled through the CEGIO approach and using ANSI-C code, with the directive *ASSUME*, which represents the associated constraints and search space. The use of such directive (*e.g.*, `__ESBMC_assume()`) is illustrated in the code fragment as shown in Figure 1.

(ii) **Specification.** This step consists in describing system behavior and properties to be checked, which results in a file according to the method proposed by Araújo *et al.* [9,10]. Indeed, the property specification is stated with *ASSERT* directives, which are used to check satisfiability and to control the verification procedure, *i.e.*, to search for violations of a given property, which, in the present case, consists of finding a function value that is smaller than the previous one. In summary, they represent calls to specific functions provided by the verification engine and also entry points for the proposed optimization. The mentioned resulting file contains the modeling and properties to be checked, as shown in Figure 1 for the function *adjiman*. In this example, ESBMC is used as verification engine, where `__ESBMC_assume()` restricts the state-space, according to the performed modeling, and `__ESBMC_assert()` checks properties.

(iii) **Verification.** Finally, the C code generated in step 2 is checked by the underlying verifier, which can return "verification successful" or "verification failed". When "verification successful" is obtained, it means that the code is correct and no property has been violated; otherwise, "verification failed" indicates that the verification engine has found a violation, *i.e.*, a value smaller than the previous one, for a particular target function. It is worth noticing that when a violation is found, the associated counterexample, which is usually provided by such tools, already indicates a smaller value. As a consequence, this new limit can then be used for updating the respective variable (*i.e.*, f_i in the example shown in Figure 1), which might iteratively lead to a minimum.

Araújo *et al.* [10] proposed three algorithms for the specification stage, which are suitable for different situations: the Generalized Algorithm (CEGIO-G), the Simplified Algorithm (CEGIO-S), and the Fast Algorithm (CEGIO-F). Figure 1 shows the specification for the function *adjiman* in CEGIO-G format, which can also be applied to any function class (*i.e.*, convex and non-convex ones). CEGIO-S is suitable for functions about which we have some prior knowledge (*e.g.*, semi- and positive-definite functions) and uses that to generate several properties in the specification step, which will be checked by the underlying verifier, with potentially increased chance of violation and reduced optimization times. Finally,

```
#define p 1
#include "math2.h"
int nondet_int();
float nondet_float();
int main() {
    float f_i = 69;
    int x[2], i;
    float X[2];
    float fobj;
    int lim[4] = {-1*p, 2*p, -1*p, 1*p};
    for (i = 0; i<2; i++){
        x[i] = nondet_int();
        X[i] = nondet_float();
    }
    for (i = 0; i<2; i++){
        __ESBMC_assume((x[i]>=lim[2*i])&&(x[i]<=lim[2*i+1]));
        __ESBMC_assume( X[i] == (float) x[i]/p );
    }
    fobj = cos2(X[0])*sin2(X[1]) - (X[0]/(X[1]*X[1]+1));
    __ESBMC_assume(fobj<f_i);
    assert(fobj>f_i);
    return 0;
}
```

Fig. 1. C code after the specification step for the function *adjiman*.

CEGIO-F can be applied to convex functions and uses their properties to restrict the associated state-space, according to the results presented by Araújo *et al.* [10], which show considerable improvement regarding optimization times. Each algorithm follows a fixed structure, which changes regarding only variable values.

3 OptCE: A Counterexample-Guided Inductive Optimization Solver

OptCE can be regarded as a front-end for model checkers that process C programs through CEGIO, where decision variables, which are in charge of generating the smallest value of a function, are checked. Such a tool can be called from a shell, via command line, and is able to optimize convex and non-convex functions, where users only need to describe, in a file, the specification and constraints regarding them, through a few code lines. In summary, OptCE is based on the CEGIO technique, which allows the discovery of global minima, while other techniques are usually trapped by local ones.

3.1 OptCE Architecture

As shown in Figure 2, users need to provide an input *.func* file (*cf. Sec. 3.2*) containing a function's specification and constraints: this is the modeling phase. Indeed, such a task reveals that some knowledge about the target problem is necessary, in order to provide a correct basis.

The first step is the specification, which receives an input file and the desired settings for optimization, such as *verifier*, *solver*, *algorithm type*, and *precision*. In Figure 2, α represents the number of decimal places of a solution, which is

indicated by the user. Based on the provided inputs, OptCE generates a specification file in ANSI-C (*cf. Figure 1*), named as `min_<function>.c`.

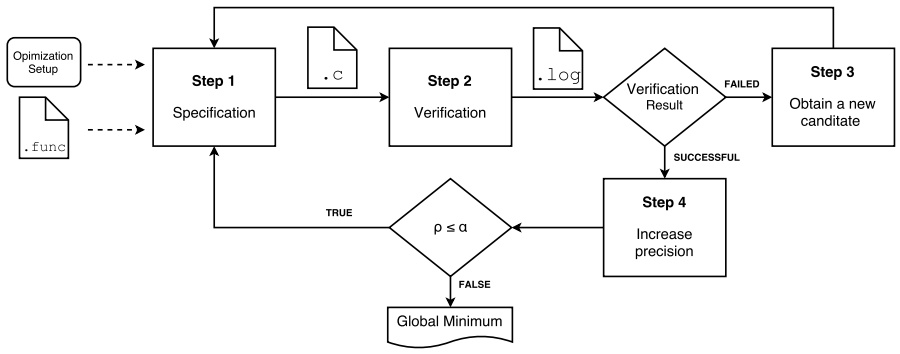

Fig. 2. An overview of the proposed OptCE architecture.

During the first execution of Step 1, ρ, which is used to establish the solution accuracy throughout the optimization process, is initialized with zero, which indicates an optimization with integer precision solutions only (*i.e.*, no decimal digits are considered). In addition, an arbitrary minimum candidate is also considered, which is actually the algorithm's initialization value and can be provided by the user with the flag `--start-value`; otherwise, it is randomly generated, which is performed during the specification step.

During Step 2, the verification task occurs, *i.e.*, the ANSI-C file with a function's specification is checked by a verification engine, whose main output is a log file with the respective verification result. If "verification failed" is obtained, that means the underlying verification engine detected a property violation through the inserted assertions and consequently generated a counterexample. In the CEGIO context, a property violation indicates that the minimum candidate is not the global minimum for that value of ρ and then the tool flow proceeds to Step 3.

In Step 3, a `.log` file with the respective counterexample is used to obtain new decision variables, which provide a new global minimum candidate lower than the previous one, *i.e.*, the initialization value or the minimum candidate of the last OptCE iteration. Then, the new minimum candidate value is obtained (*i.e.*, extracted and computed from the counterexample), and used to perform Step 1 again, starting a new iteration and generating a new specification file. Such a procedure is iteratively performed until the verification step (Step 2) returns a `.log` file with "verification successful", which means that there are no decision variables capable of finding a minimum value smaller than the current one, considering the current value of the precision variable (ρ). When the verification result is "verification successful", OptCE proceeds to Step 4.

In Step 4, ρ is increased by one, *i.e.*, the precision associated to the optimal solution is increased by one decimal place, which is followed by a check that evaluates whether it is smaller than or equal to the desired accuracy (indicated by the user). If ρ is larger than α (the condition $\rho \leq \alpha$ is false), OptCE has found the solution (global minimum), considering the desired precision; otherwise, the OptCE's general flow (Steps 1-3) is repeated with the updated precision ρ, *i.e.*, the algorithm returns to Step 1 and generates a new specification file.

3.2 Input File for OptCE

The present input file consists of two parts: function specification and associated constraints, which are separated by a character "#" isolated in a row. At the top of an input file, a function must be described with an ANSI-C variable assignment ending with ";" and using variable `fobj` that represents the objective function (see Figure 3). We summarize the OptCE input language in Figure 3.

$$
\begin{aligned}
\mathit{Fml} &::= \mathit{Var} \mid \mathit{true} \mid \mathit{false} \mid \mathit{Fml} \wedge \mathit{Fml} \mid \ldots \mid \mathit{Exp} = \mathit{Exp} \mid \ldots \\
\mathit{Exp} &::= \mathit{Var} \mid \mathit{Const} \mid \mathit{Var}[\mathit{Exp}] \mid \mathit{Var}[\mathit{Exp}][\mathit{Exp}] \mid \mathit{Exp} + \mathit{Exp} \mid \ldots \\
\mathit{Cmd} &::= \mathit{Var} = \mathit{Exp} \mid \mathit{Var} = * \mid \mathit{Fml} \mid \mathit{sin2}(\mathit{Var}) \mid \mathit{cos2}(\mathit{Var}) \\
&\quad \mid \mathit{floor2}(\mathit{Var}) \mid \mathit{sqrt2}(\mathit{Var}) \mid \mathit{abs2}(\mathit{Var}) \\
\mathit{Prog} &::= \mathit{Cmd}; \ldots; \#\mathit{Cmd};
\end{aligned}
$$

Fig. 3. OptCE input program language.

Eq. (3) presents the format adopted for constraint matrices, where the associated number of lines indicates the amount of decision variables and columns 1 and 2 represent the lower and upper bounds, respectively.

$$
\begin{bmatrix}
x_{11} & x_{12} \\
x_{21} & x_{22} \\
\multicolumn{2}{c}{\cdots} \\
x_{n1} & x_{n2}
\end{bmatrix} \tag{3}
$$

The constraints of the considered optimization problem (Eq. (2)) can be represented by $A = [\ -1\ 2\ ; -1\ 1]$ and an input file containing the entire optimization problem (related to Eq. (2)) is illustrated in Figure 4.

```
fobj = cos2(x1)*sin2(x2) - (x1/(x2*x2+1));
#
A = [-1  2; -1  1];
```

Fig. 4. Input file for function *adjiman*.

3.3 OptCE Features

The current OptCE version allows us to define different configurations regarding the optimization process (*i.e.*, optimization algorithm and verification engine), which is used to reduce optimization times. Thus, users have to add suitable flags during a call via command-line. The following configurations are supported:

- **BMC configuration:** chooses between model checkers CBMC (`--cbmc`) and ESBMC (`--esbmc`);
- **Solver configuration:** chooses between solvers Boolector (`--boolector`), Z3 (`--z3`), MathSAT (`--mathsat`), and MiniSAT (`--minisat`);
- **Algorithm configuration:** chooses between the proposed algorithms, where the flag `--generalized` implements the CEGIO-G algorithm (*cf. Sec. 2*), which is used when there is no prior knowledge about the objective function, the flag `--positive` implements the CEGIO-S algorithm (*cf. Sec. 2*), which is used when a function is semi- and positive-definite, and the flag `--convex` implements the CEGIO-F algorithm (*cf. Sec. 2*), which is used for convex functions.
- **Initialization:** assigns an initial minimum candidate value (`--start-value` =*value*), which is random by default;
- **Insert library:** users can include their own library containing implementations of operators and functions used in the objective function's description (`--library`=*name-library*);
- **Timeout:** configures the time limit, in seconds (`--timeout`=*value*).
- **Precision:** sets the desired precision, *i.e.*, the number of decimal places of a solution (`--precision` =*value*).

3.4 Optimizing via OptCE

The user must create a description input file to find the global minimum of a function using the OptCE tool, as explained in subsection 3.2. Figure 5 shows all possible OptCE calls with input file and set of properties. Here, we employ the function *adjiman* to illustrate the use of OptCE, considering the input file shown in Figure 4.

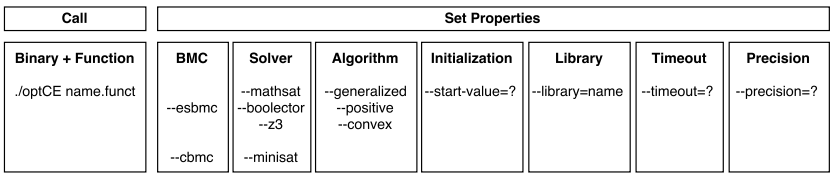

Fig. 5. OptCE configuration options.

Currently, OptCE supports two verifiers: CBMC [13] and ESBMC [21]. Optimization employing CBMC as model checker (`-cbmc`) uses MiniSAT as default

solver, while ESBMC (`--esbmc`) uses MathSAT. In our evaluation, we also tried to use the SMT solvers available in CBMC, but it failed to check all benchmarks reported in Table 1 due to problems in the SMT back-end. Regarding ESBMC, the user can choose between solvers Z3 (`--z3`) or Boolector (`--boolector`); however, we did not further evaluate other SMT solvers (*e.g.*, CVC4 and Yices). Indeed, verification times vary according to the selected verifier and solver and, as already mentioned, the user has the possibility to choose different configurations. If a given user is unsure about which verifier and solver to select, then a default choice would be to employ ESBMC with MathSAT or CBMC with MiniSAT, given that they normally present the shortest execution times; however, our experimental evaluation does not conclusively show that they are the best possible configurations (given the small benchmark set). As future work, we intend to automatically select the verifier and solver pair, using machine learning techniques that take into account objective functions, with a large set of benchmarks. Indeed, such an approach is similar to the work done by Hutter *et al.* [22], who apply a parameter optimization tool to improve SAT solvers for large, real-world bounded model-checking instances, via automatic tuning of decision procedures.

Another important parameter is the algorithm type, which can be `--convex`, for convex functions, `--positive`, for semi- and definite-positive functions, and `--generalized`, for functions about which we do not have any prior knowledge. Since Eq. (1) is not convex and it is not possible to ensure that it is non-negative, the suggested setup uses flag `--generalized` (*./optCE adjiman.func* `--generalized`).

Following the execution flow illustrated in Figure 5, the flag `--start-value` is used to specify the proposed algorithm's initialization (*./optCE <name>.func* `--start-value=20`) and, when it is not adopted, such a value is assigned in a random way. We noticed that variations regarding initialization values do not significantly influence convergence times, since OptCE evaluates only the integer part of the solutions, at the beginning of the optimization tasks. In addition, checking with integer values is fast, it is normal to get "verification failed" in the first round, and a "verification failed" result is generally faster than a "verification successful" one, as also experimentally observed by Araújo *et al.* [9,10].

If the input function consists of arithmetic operators, then it is not mandatory to use the flag `--library`; however, when mathematical functions are present, it is necessary to implement them in ANSI-C. Such implementations considerably influence the verification results and the simpler they are, *i.e.*, the smaller their number of operations and loops is, the easier it is for the proposed approach to conclude the verification tasks. In the case of the *adjiman* function, which uses mathematical functions such as *sin()* and *cos()*, the library *math2.h* was created, with our own implementation, which was included using the flag `--library` (*./optCE adjiman.func* `--library=math2.h`). This library contains an improved implementation of the original *math.h*, which includes pre- and post-conditions to ensure that a (given) predicate holds before and after the execution of a (given) math function, respectively.

Our mathematical functions in *math2.h* have the same name of the corresponding elements in the ANSI-C library, except that we appended the character 2 (*e.g.*, *cos2()*, *sin2()*, *abs2()*).

The `--timeout` flag is used to interrupt optimization processes, if they reach the indicated time limit (*./optCE <name_function>.func* `--timeout=3600`). Finally, the user has the option to define the OptCE's solution accuracy, *i.e.*, the `--precision` flag indicates the number of decimal places of a solution. When a reference value is not provided, OptCE finds a global minimum with 3 decimal places, by default.

4 Experimental Evaluation

This section reports the performed experiments configuration and execution, along with an analysis of the results obtained with OptCE.

4.1 Experimental Objectives

Our experiments have been carried out seeking answers to the following questions:

RQ1 **(correctness)** Is OptCE able to find the global minima of functions?
RQ2 **(sanity check)** Does the settings choice between BMC tools and solvers influence optimization results?
RQ3 **(performance)** What are the advantages and disadvantages of OptCE, in comparison with traditional optimization techniques?

4.2 Description of the Benchmarks

In order to evaluate the proposed tool and answer those research questions presented in section 4.1, a benchmark suite with 10 convex and nonconvex functions was created, with functions related to optimization problems extracted from the available literature [19]. They have different characteristics: continuous, differentiable, separable, non-separable, scalable, non-scalable, uni-modal, and multi-modal, including sine, cosine, polynomials, floor, sum, and square root. The chosen benchmarks are shown in Table 1, as follows: benchmark name, optimization domain, and global minimum.

All functions were used to evaluate the flag `--generalized`, which implements the CEGIO-G algorithm (*cf. Sec. 2*). In order to evaluate the flag `--positive`, which implements the CEGIO-S algorithm (*cf. Sec. 2*), semi-definite positive functions *Booth*, *Himmelblau*, and *Leon* were used. Lastly, functions *Zettl*, *Rotated Ellipse*, and *Sum Square* were used to evaluate the flag `--convex`, which implements the CEGIO-F algorithm (*cf. Sec. 2*).

The results of the proposed approach were compared with the ones presented by other techniques (*i.e.*, genetic algorithm, particle swarm, pattern search, simulated annealing, and nonlinear programming), where all benchmarks were executed with the MATLAB's optimization toolbox (2016b) [23].

Table 1. Benchmark Suite.

#	Benchmark	Domain	Global Minimum
1	Alpine 1	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
2	Cosine	$-1 \leq x_i \leq 1$	$f(0,0) = -0.2$
3	Styblinski Tang	$-5 \leq x_i \leq 5$	$f(2.903, 2.903) = -78.332$
4	Zirilli	$-10 \leq x_i \leq 10$	$f(1.046, 0) \approx -0.3523$
5	Booth	$-10 \leq x_i \leq 10$	$f(1,3) = 0$
6	Himmeblau	$-5 \leq x_i \leq 5$	$f(3,2) = 0$
7	Leon	$-2 \leq x_i \leq 2$	$f(1,1) = 0$
8	Zettl	$-5 \leq x_i \leq 10$	$f(0.029, 0) = -0.0037$
9	Sum Square	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
10	Rotated Ellipse	$-500 \leq x_i \leq 500$	$f(0,0) = 0$

Similarly to the experiments performed by Araújo *et al.* [9,10], the elapsed times presented in the following tables are related to the average CPU time measured with the *times* system call (POSIX system) of 20 consecutive executions for each benchmark, where the measurement unit is always in seconds. Finally, our experiments were set for obtaining the global minima with 3 decimal places and were conducted on an otherwise idle computer equipped with Intel Core i7-4790 CPU 3.60 GHz, 16 GB of RAM, and Linux OS Ubuntu 14.10.

4.3 Experimental Results

The experimental results are presented in four tables. Tables 2, 3, and 4 refer to the benchmarks with the flags `--generalized`, `--positive`, and `--convex`, respectively. Table 5 refers to a comparison between OptCE v1.0 and other traditional techniques. Each column of Table 2 is described as follows: column 1 is related to functions of the reference benchmark suite, columns 2, 3 and 4 are related to the ESBMC v3.1.0 configuration with MathSAT v.5.3.13, Z3 v4.5.0, and Boolector v2.2.0 solvers, respectively, and column 5 is related to the CBMC v4.5 configuration with MiniSat v2.2.0.

Table 2. Execution times for the generic algorithm (CEGIO-G [10]), in seconds.

#	ESBMC			CBMC
	MathSAT	Z3	Boolector	MiniSAT
1	**1068**	105192	3387	5344
2	**4130**	80481	5003	8509
3	**443**	37778	2027	2438
4	468	387	**190**	1143
5	7	1244	4016	**2**
6	12	14205	6217	**4**
7	5	2443	212	**2**
8	13	753	389	**9**
9	18	4171	4438	**13**
10	3	72	39	**2**

The overall minimum was found in all benchmarks, considering all combinations between BMC tools and solvers. As presented by Araújo *et al.* [10],

those algorithms ensure the global minimum, considering the desired accuracy, which was described in previous section; their proofs of convergence are provided in [10], which confirm the experimental results. The optimization times varied significantly in Table 2, which makes it difficult to reason about the best configuration; however, according to Figure 6, the total optimization time with the configuration ESBMC + MathSAT (the best one) was 2.8 times faster than the one presented by CBMC + MiniSAT, while the configuration ESBMC + Z3 presented the longest execution time, being 40 times longer than the best case.

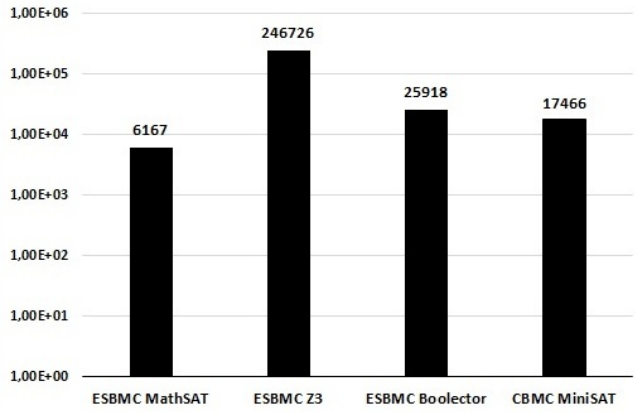

Fig. 6. Histogram of the total optimization time for the adopted benchmark suite, in logarithmic scale.

Another interesting observation regarding Table 2 is that although CBMC + MiniSAT provided the second best performance, considering the entire benchmark suite, such a configuration was the best in 60% of the benchmarks, *i.e.*, a few cases required long run times, but they were exceedingly time consuming. Benchmarks $\#1-4$ are non-convex and presented long times when searching for the global minima, considering all possible settings. Benchmarks $\#5-7$ are semi- and positive-definite functions, while $\#8-10$ are convex ones. Regarding them, OptCE was able to find solutions using the algorithms CEGIO-S and CEGIO-F, by providing, respectively, the flags `--positive` and `--convex`.

Nonetheless, in order to evaluate the implementation of CEGIO-G, all benchmarks were optimized with the flag `--generalized`. The experiments were repeated for different combinations of model checkers and SAT/SMT solvers, *i.e.*, ESBMC was combined with three solvers (MathSAT, Z3, and Boolector) and CBMC with MiniSAT only. Particularly, the combinations ESBMC + MathSAT and CBMC + MiniSAT presented results significantly better than the ones provided by other configurations of OptCE, given that Boolector does not support floating-point arithmetic [17]. In particular, MathSAT (the one that obtained the best results) supports both fixed- and floating-point arithmetic and,

surprisingly, the performance for floating-point optimization is significantly better if compared to the fixed-point one. As a consequence, when using the flag `--generalized`, the configurations ESBMC + MathSAT and CBMC + MiniSAT are recommended.

Table 3 presents the results for the flag `--positive`, which is suitable for semi- and positive-definite functions. As a consequence, we used only benchmarks $\#5-7$, in this experiment. Those functions make use of modules with high even powers, *i.e.*, by mathematical inspection we can ensure that such functions can not reach one global negative minimum. Table 3 compares the use of the flags `--generalized` and `--positive`, for this class of problems. One may notice that the implementation of the CEGIO-S algorithm with the flag `--positive` does indeed work, since optimization times were significantly reduced, if compared to the flag `--generalized`, in all possible configurations. This happens because the solution search space is reduced, by ignoring the negative part.

Table 3. Execution times for the positive algorithm (CEGIO-S [10]), in seconds.

#	`--positive`				`--generalized`			
	ESBMC			CBMC	ESBMC			CBMC
	MathSAT	Z3	Boolector	MiniSAT	MathSAT	Z3	Boolector	MiniSAT
5	3	**<1**	1	3	7	1244	4016	2
6	4	**1**	**1**	2	12	14205	6217	4
7	3	**<1**	1	2	5	2443	212	2

The CEGIO-F algorithm implementation is assigned with the flag `--convex`. In order to evaluate its performance, benchmarks $\#8-10$ were used, because they are convex functions, and their results are presented in Table 4. The optimization times using a specific algorithm for this function class were considerably lower than the times presented by the generalized algorithm. That happens because, in this algorithm and with each performed check, the search space is reduced according to the found global minimum candidate, which then decreases verification and, consequently, optimization times.

Table 4. Execution times for the convex algorithm (CEGIO-F [10]), in seconds.

#	`--convex`				`--generalized`			
	ESBMC			CBMC	ESBMC			CBMC
	MathSAT	Z3	Boolector	MiniSAT	MathSAT	Z3	Boolector	MiniSAT
8	15	6	21	**5**	13	753	389	9
9	14	**3**	19	5	18	4171	4438	13
10	3	**1**	2	2	3	72	39	2

The best results using the proposed tool, for each benchmark, are presented in Table 5, along with results for other techniques. The *configuration* column shows the combinations regarding algorithm types (by the initials of flags "G" for `--generalized`, "P" for `--positive`, and "C" for `--convex`), BMC tools, and

solvers. The comparison is performed with traditional optimization techniques: genetic algorithm (GA), particle swarm (ParSwarm), pattern search (PatSearch), simulated annealing (SA), and nonlinear programming (NLP). All evaluated benchmarks were executed 1000 times with the traditional techniques, using MATLAB, and 20 times with CEGIO, using OptCE. The number of repetitions was selected to ensure the convergence of hit rate for all algorithms.

Table 5. Experimental results for traditional techniques and the best proposed CEGIO algorithms, in seconds.

#	OptCE			GA		ParSwarm		PatSearch		SA		NLP	
	Configuration	R%	T	R%	T	R%	T	R%	T	R%	T	R%	T
1	G + ESBMC + MathSAT	100	1068	29.1	1	22.2	3	16	4	0.4	1	4.8	9
2	G + ESBMC + MathSAT	100	4130	100	9	9.8	1	96.7	3	88.5	2	28.4	2
3	G + ESBMC + MathSAT	100	443	68.1	9	47.8	1	51.8	3	99.5	1	35.8	2
4	G + ESBMC + Boolector	100	190	95.7	9	53.9	1	98.8	3	74.4	1	62.5	2
5	P + ESBMC + Z3	100	< 1	100	10	100	2	100	6	93.5	1	100	2
6	P + ESBMC + Z3	100	1	42.4	9	43.9	1	26	3	21	1	35	2
7	P + ESBMC + Z3	100	< 1	84.4	1	80.3	2	1	7	24.3	1	100	4
8	C + CBMC + MiniSAT	100	5	100	9	48.1	1	99.8	4	26.4	1	100	3
9	C + ESBMC + Z3	100	3	100	9	71.5	1	100	4	96.9	1	100	2
10	C + ESBMC + Z3	100	1	100	9	100	2	100	7	99.8	1	100	2

OptCE's hit rate is 100% for this benchmark suite, considering the domain established in Table 1, for each benchmark. The present experiments show that OptCE generally takes longer than other techniques, in order to locate the global minima; however, its hit rate is always higher. In particular, the time results with the flags `--positive` (CEGIO-S) and `--convex` (CEGIO-F) are similar to what is provided by the other techniques, but with superior hit rates. The chosen traditional optimization techniques, in many cases, failed to obtain solutions for the adopted benchmarks, considering the established precision of 3 decimal places. That happened because they are sensitive to non-convexity and, in many cases, they get trapped by local minima, which resulted in sub-optimal solutions. If only benchmarks 8, 9, and 10, in Table 1, are evaluated, which are convex functions, the rate obtained by existing methods is 100%, since those functions do not have local minima that can compromise their results.

In summary, the proposed technique can be used in any optimization problem, but there are always restrictions regarding the time and number of variables. Usually, cost functions in practical problems are distance or power functions, *i.e.*, they are semi- and positive-definite. Therefore, as OptCE has the CEGIO-S algorithm implemented in its structure, which is specific to this function class, it implies that OptCE is able to solve those particular optimization problems.

OptCE presents good performance with non-convex functions, if compared to the traditional techniques, because the global minima are found in all benchmarks. Traditional techniques, in turn, are lost at local minimum and return sub-optimal solutions, which then reduces their hit rate.

The performance of OptCE using specific flags for convex and positive-definite functions proved to be competitive, once the obtained execution times were very close to the ones from other techniques, given that global minima were found in all cases. Depending on the problem type, the number of solution decimal places might be lower than the amount used in this experimental evaluation. For those cases, execution times regarding the location of optimal solutions are reduced, once there are fewer decimal places to check, which then implies fewer verifications and fewer states to be considered.

5 Related Work

Since the earliest research with SMT application to solve optimization problems, which was presented by Nieuwenhuis and Oliveras [24], several satisfiability-theory based tools have emerged, with the purpose of solving optimization problems. Conversely, various SAT/SMT specialized solvers have been developed, which employ optimization techniques in their engines to improve solving performance (*e.g.*, `ABsolver` [25], and `CalCs` [26]). Shoukry *et al.* [27] proposed the Satisfiability Modulo Convex (SMC) Optimization [28] to solve satisfiability problems over SMC formulas, which generalizes several formulas over Boolean and nonlinear real arithmetic.

Recently, νZ [29] extended the SMT solver Z3 [16] for linear optimization problems and Li *et al.* proposed the `SYMBA` algorithm [30], which is an SMT-based symbolic optimization algorithm that uses linear real arithmetic theory and SMT solvers, as black boxes. Similarly, `OptiMathSat` presented by Sebastiani and Trentin [31] is also an optimization tool that extends MathSAT5 SMT solver to allow solving linear functions in the boolean, rational, and integer domains, or a combination of them. Although the OptCE tool presented in this study is based on satisfiability theories, it does not employ SAT/SMT solvers directly, in contrast to other techniques [29,30,31]. OptCE incorporates the model checking approach and employs SAT- and SMT-based model checkers to model, specify, and verify ANSI-C representations of optimization problems by exploiting the counterexample provided by them.

Model-checking has already been employed to model and solve optimization problems, in some previous studies. Trindade *et al.* [32,33] used the ESBMC tool to solve optimization problems over booleans decision variables related to hardware/software partition, in embedded systems. Araújo *et al.* [9,10] proposed the CEGIO algorithms to globally optimize non-convex functions on the rational domain, with adjustable precision.

Most previous studies related to SMT-based optimization can only solve linear problems over integer, rational, and Boolean domains, in specific cases. Indeed, only a few studies [27] are able to solve non-linear problems, but they are also constrained to convex functions. In contrast, this paper proposes a new tool that implements the CEGIO algorithms [9,10] and is able to globally minimize a wide variety of functions: linear or non-linear, convex or non-convex, and continuous or discontinuous.

6 Conclusion

OptCE is a novel optimization tool that models a wide range of constrained optimization problems (convex, nonlinear, and nonconvex) as a model checking problem and inductively analyzes counterexamples, in order to achieve global optimization of functions, by employing SAT- or SMT-based verification. In particular, this tool is based on a class of optimization algorithms, named CEGIO, and it is able to ensure the global optimal convergence with a given precision. OptCE supports the following features: three different CEGIO algorithms (CEGIO-G, CEGIO-S, and CEGIO-F), two state-of-art BMC tools (CBMC and ESBMC), and four SAT/SMT solvers (MiniSAT, Boolector, Z3, and MathSAT).

Our experiments showed that OptCE achieved 100% of hit rate, being able to ensure the global optimization. In contrast, other traditional techniques (GA, PatSearch, ParSwarm, NLP, and SA) employed for comparison were usually trapped by local minima. In addition, the experimental results indicated that the most flexible CEGIO algorithm (CEGIO-G), which is suitable for every function class supported by OptCE, presented times significantly longer than the others from the CEGIO algorithms and traditional techniques, despite ensuring the global optimization. Nonetheless, the other two CEGIO algorithms (CEGIO-S and CEGIO-F), which are suitable for nonnegative and convex optimization problems, respectively, were able to solve global optimization problems with times similar to the ones provided by the traditional techniques, but with superior hit rate.

OptCE is available for free download (Linux x86 version),[1] including documentation, benchmarks, results, publications, and source code. Although the OptCE's time performance is slow, it has been and will be continuously improved, given that verifiers and SAT/SMT solvers evolve, even with the inclusion of new and adaptive techniques, such as machine learning [22]. Future work includes parallelization and state space partitioning, thus linearly reducing checking times. We also intend to enhance our model-checking procedure for reducing the verification time by means of automatic invariant generation [34,35].

Acknowledgements

This research was supported by FAPEAM and CNPq. Higo Albuquerque was also supported by a CAPES studentship.

References

1. H. Park, P. Bradley, P. Greisen Jr., Y. Liu, V. K. Mulligan, D. E. Kim, D. Baker, F. DiMaio, Simultaneous optimization of biomolecular energy functions on features from small molecules and macromolecules, J. Chem. Theory Comput. 12 (12) (2016) 6201–6212.

[1] Available at `http://esbmc.org/benchmarks/optce.zip`

2. K. D. Cooper, L. Torczon, Engineering a Compiler, Morgan Kaufmann, San Francisco, CA, USA, 2004.
3. K. Deb, Optimization for Engineering Design: Algorithms and Examples, Prentice-Hall of India, 2004.
4. K. Vergidis, A. Tiwari, B. Majeed, Business process analysis and optimization: Beyond reengineering, IEEE Trans. Syst., Man, Cybern., C Appl Rev 38 (1).
5. P. J. M. Laarhoven, E. H. L. Aarts (Eds.), Simulated Annealing: Theory and Applications, Kluwer Academic Publishers, Norwell, MA, USA, 1987.
6. A. Olsson, Particle Swarm Optimization: Theory, Techniques and Applications, Engineering tools, techniques and tables, Nova Science Publishers, 2011.
7. D. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Artificial Intelligence, Addison-Wesley Publishing Company, 1989.
8. C. Floudas, Deterministic Global Optimization, Nonconvex Optimization and Its Applications, Springer, 2000.
9. R. Araújo, I. Bessa, L. Cordeiro, J. E. C. Filho, SMT-based verification applied to non-convex optimization problems, in: SBESC, 2016, pp. 1–8.
10. R. Araújo, I. Bessa, L. Cordeiro, J. E. C. Filho, Counterexample guided inductive optimization, in: arXiv:1704.03738 [cs.AI], 2017, pp. 1–32.
URL `http://arxiv.org/abs/1704.03738`
11. R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, A. Udupa, Syntax-guided synthesis, in: FMCAD, 2013, pp. 1–8.
12. A. Solar-Lezama, The sketching approach to program synthesis, in: APLAS, Vol. 5904 of LNCS, 2009, pp. 4–13.
13. D. Kroening, M. Tautschnig, CBMC - C bounded model checker - (competition contribution), in: TACAS, Vol. 8413 of LNCS, 2014, pp. 389–391.
14. L. Cordeiro, B. Fischer, J. Marques-Silva, SMT-based bounded model checking for embedded ANSI-C software, IEEE TSE 38 (4) (2012) 957–974.
15. A. Cimatti, A. Griggio, B. Schaafsma, R. Sebastiani, The MathSAT5 SMT Solver, in: TACAS, Vol. 7795 of LNCS, 2013, pp. 93–107.
16. L. De Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: TACAS, Vol. 4963 of LNCS, 2008, pp. 337–340.
17. R. Brummayer, A. Biere, Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays, in: TACAS, Vol. 5505 of LNCS, 2009, pp. 174–177.
18. N. Eén, N. Sörensson, An Extensible SAT-solver, in: SAT, Vol. 2919 of LNCS, 2003, pp. 502–518.
19. M. Jamil, X. Yang, A literature survey of benchmark functions for global optimization problems, CoRR abs/1308.4008.
URL `http://arxiv.org/abs/1308.4008`
20. C. Baier, J.-P. Katoen, Principles of Model Checking (Representation and Mind Series), The MIT Press, 2008.
21. J. Morse, M. Ramalho, L. C. Cordeiro, D. Nicole, B. Fischer, ESBMC 1.22 - (Competition Contribution), in: TACAS, Vol. 8413 of LNCS, 2014, pp. 405–407.
22. F. Hutter, D. Babic, H. H. Hoos, A. J. Hu, Boosting verification by automatic tuning of decision procedures, in: FMCAD, 2007, pp. 27–34.
23. The Mathworks, Inc., Matlab Optimization Toolbox User's Guide (2016).
24. R. Nieuwenhuis, A. Oliveras, On SAT Modulo Theories and Optimization Problems, in: SAT, Vol. 4121 of LNCS, 2006, pp. 156–169.
25. A. Bauer, M. Pister, M. Tautschnig, Tool-support for the analysis of hybrid systems and models, in: DATE, 2007, pp. 924–929.

26. P. Nuzzo, A. A. A. Puggelli, S. A. Seshia, A. L. Sangiovanni-Vincentelli, CalCS: SMT solving for non-linear convex constraints, Tech. Rep. UCB/EECS-2010-100, EECS Department, University of California, Berkeley (Jun 2010).
27. Y. Shoukry, P. Nuzzo, I. Saha, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, P. Tabuada, Scalable lazy smt-based motion planning, in: CDC, 2016, pp. 6683–6688.
28. Y. Shoukry, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, P. Tabuada, SMC: Satisfiability Modulo Convex Optimization, in: HSCC, 2017, pp. 19–28.
29. N. Bjørner, A. Phan, L. Fleckenstein, νz - an optimizing SMT solver, in: TACAS, Vol. 9035 of LNCS, 2015, pp. 194–199.
30. Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, M. Chechik, Symbolic Optimization with SMT Solvers, in: POPL, 2014, pp. 607–618.
31. R. Sebastiani, P. Trentin, Optimathsat: A tool for optimization modulo theories, in: CAV, Vol. 9206 of LNCS, 2015, pp. 447–454.
32. A. Trindade, H. Ismail, L. Cordeiro, Applying multi-core model checking to hardware-software partitioning in embedded systems, in: SBESC, 2015, pp. 102–105.
33. A. Trindade, L. Cordeiro, Applying SMT-based verification to hardware/software partitioning in embedded systems, DES AUTOM EMBED SYST 20 (1) (2016) 1–19.
34. H. Rocha, H. Ismail, L. Cordeiro, R. Barreto, Model Checking Embedded C Software Using k-Induction and Invariants, in: SBESC, 2015, pp. 90–95.
35. M. Y. R. Gadelha, H. I. Ismail, L. C. Cordeiro, Handling loops in bounded model checking of C programs via k-induction, STTT 19 (1) (2017) 97–114.