Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories

Rodrigo F. Araújo[a], Higo F. Albuquerque[b], Iury V. de Bessa[b], Lucas C. Cordeiro[c], João Edgar C. Filho[b]

[a]*Federal Institute of Amazonas, Campus Manaus Distrito Industrial, Avenida Governador Danilo Areosa, 1672, Distrito Industrial, 69075351, Manaus, AM, Brasil*
[b]*Department of Electricity, Federal University of Amazonas, Avenida General Rodrigo Octávio, 6200, Coroado I, Manaus, AM, 69077-000*
[c]*Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, OXFORD, OX1 3QD, UK*

Abstract

This paper describes three variants of a counterexample guided inductive optimization (CEGIO) approach based on Satisfiability Modulo Theories (SMT) solvers. In particular, CEGIO relies on iterative executions to constrain a verification procedure, in order to perform inductive generalization, based on counterexamples extracted from SMT solvers. CEGIO is able to successfully optimize a wide range of functions, including non-linear and non-convex optimization problems based on SMT solvers, in which data provided by counterexamples are employed to guide the verification engine, thus reducing the optimization domain. The present algorithms are evaluated using a large set of benchmarks typically employed for evaluating optimization techniques. Experimental results show the efficiency and effectiveness of the proposed algorithms, which find the optimal solution in all evaluated benchmarks, while traditional techniques are usually trapped by local minima.

Keywords:
Satisfiability Modulo Theories (SMT), Model checking, Global optimization, Non-convex optimization

1. Introduction

Optimization is an important research topic in many fields, especially in computer science and engineering [1]. Commonly, scientists and engineers have to find parameters, which optimize the behavior of a given system or the value of a given function (*i.e.*, an optimal solution). Optimization characterizes and distinguishes the engineering gaze over a problem; for this particular reason, previous studies showed that optimization is one of the main differences between engineering design and technological design [2].

Computer science and optimization maintain a symbiotic relationship. Many important advances of computer science are based on optimization theory. As example, planning and decidability problems (*e.g.*, game theory [3]),

resource allocation problems (*e.g.*, hardware/software co-design [4]), and computational estimation and approximation (*e.g.*, numerical analysis [5]) represent important optimization applications. Conversely, computer science plays an important role in recent optimization studies, developing efficient algorithms and providing respective tools for supporting model management and results analysis [6].

There are many optimization techniques described in the literature (*e.g.*, simplex [7], gradient descent [8], and genetic algorithms [9]), which are suitable for different classes of optimization problems (*e.g.*, linear or non-linear, continuous or discrete, convex or non-convex, and single- or multi-objective). These techniques are usually split into two main groups: deterministic and stochastic optimization. Deterministic optimization is the classic approach for optimization algorithms, which is based on calculus and algebra operators, *e.g.*, gradients and Hessians [10]. Stochastic optimization employs randomness in the optima search procedure [10]. This paper presents a novel class of search-based optimization algorithm that employs non-deterministic representation of decision variables and constrains the state-space search based on counterexamples produced by an SMT solver, in order to ensure the complete global optimization without employing randomness. This class of techniques is defined here as *counterexample guided inductive optimization* (CEGIO), which is inspired by the syntax-guided synthesis (SyGuS) to perform inductive generalization based on counterexamples provided by a verification oracle [11].

Particularly, a continuous non-convex optimization problem is one of the most complex problems. As a result, several traditional methods (*e.g.*, Newton-Raphson [1] and Gradient Descent [8]) are inefficient to solve that specific class of problems [1]. Various heuristics are developed for obtaining approximated solutions to those problems; heuristics methods (*e.g.*, ant colony [12] and genetic algorithms [9]) offer faster solutions for complex problems, but they sacrifice the system's correctness and are easily trapped by local optimal solutions.

This paper presents a novel counterexample guided inductive optimization technique based on SMT solvers, which is suitable for a wide variety of functions, even for non-linear and non-convex functions, since most real-world optimization problems are non-convex. The function evaluation and the search for the optimal solution is performed by means of an iterative execution of successive verifications based on counterexamples extracted from SMT solvers. The counterexample provides new domain boundaries and new optima candidates. In contrast to other heuristic methods (*e.g.*, genetic algorithms), which are usually employed for optimizing this class of function, the present approaches always find the global optimal point.

This study extends the previous work of Araújo *et al.* [13] and presents three variants of a counterexample guided inductive optimization approach based on SMT solvers, which improve the technique performance for specific class of functions. Furthermore, the experimental evaluation is largely expanded, since the algorithms are executed for additional optimization problems and the performance of each proposed algorithm is compared to six well-known optimization techniques. The present CEGIO approaches are able to find the correct global minima for 100% of the benchmarks, while other techniques are

usually trapped by local minima, thus leading to incorrect solutions.

1.1. Contributions

Our main original contributions are:

- **Novel counterexample guided inductive optimization approach**. This work describes three novel variants of a counterexample guided inductive optimization approach based on SMT solvers: generalized, simplified, and fast algorithms. The generalized algorithm can be used for any constrained optimization problem and presents minor improvements w.r.t. Araújo *et al.* [13]. The simplified algorithm is faster than the generalized one and can be employed if information about the minima location is provided, *e.g.*, the cost function is semi-definite positive. The fast algorithm presents a significant speed-up if compared to the generalized and simplified ones, but it can only be employed for convex functions.
- **Convergence Proofs**. This paper presents proofs of convergence and completeness (omitted in Araújo *et al.* [13]) for the proposed counterexample guided inductive optimization algorithms.
- **SMT solvers performance comparison**. The experiments are performed with three different SMT solvers: Z3 [14], Boolector [15], and MathSAT [16]. The experimental results show that the solver choice can heavily influence the method performance.
- **Additional benchmarks**. The benchmark suite is expanded to 30 optimization functions extracted from the literature [17].
- **Comparison with existing techniques.** The proposed technique is compared to genetic algorithm [9], particle swarm [18], pattern search [19], simulated annealing [20], and nonlinear programming [21], which are traditional optimization techniques employed for non-convex functions.

1.2. Availability of Data and Tools

Our experiments are based on a set of publicly available benchmarks. All tools, benchmarks, and results of our evaluation are available on a supplementary web page `http://esbmc.org/benchmarks/jscp2017.zip`.

1.3. Outline

Section 2 discusses related studies. Section 3 provides an overview of optimization problems and techniques, describes background on software model checking, and error and accuracy problems due to numerical representation by computers. Section 4 describes the generalized optimization algorithm and its completeness proof. Futhermore, it describes the ANSI-C model developed for optimization problems, which is suitable for the counterexample-guided inductive optimization procedure. Section 5 describes two variants for the generalized optimization algorithm: the simplified and the fast optimization algorithm together with their respective completeness proofs. Section 6 reports the experimental results for evaluating all proposed optimization algorithms, while Section 8 concludes this work and proposes further studies.

2. Related Work

SMT solvers have been widely applied to solve several types of verification, synthesis, and optimization problems. They are typically used to check the satisfiability of a logical formula, returning assignments to variables that evaluate the formula to true, if it is satisfiable; otherwise, the formula is said to be unsatisfiable. Nieuwenhuis and Oliveras [22] presented the first research about the application of SMT to solve optimization problems, defining the so-called Optimization Modulo Theories (OMT) based on arithmetic reasoning and limited to booleand variables. More recently, Nadel and Ryvchin [23] propose a different SMT-based optimization algorithm based on bit-vector SMT theory, and so called Optimization modulo Bit-Vectors (OBV). Since then, SMT solvers have been used to solve different optimization problems, *e.g.*, minimize errors in linear fixed-point arithmetic computations in embedded control software [24]; reduce the number of gates in FPGA digital circuits [25]; hardware/software partition in embedded systems to decide the most efficient system implementation [26–28]; and schedule applications for a multi-processor platform [29]. All those previous studies use SMT-based optimization over a Boolean domain to find the best system configuration given a set of metrics. In particular, in Cotton *et al.* [29] the problem is formulated as a multi-objective optimization problem. Recently, Shoukry *et al.* [30] proposed a scalable solution for synthesizing a digital controller and motion planning for under-actuated robots from LTL specifications. Such solution is more flexible and allows solving a wider variety of problems, but they are focused on optimization problems that can be split into a Boolean part and other convex part.

In addition, there were advances in the development of different specialized SMT solvers that employ generic optimization techniques to accelerate SMT solving, *e.g.*, the `ABsolver` [31], which is used for automatic analysis and verification of hybrid-system and control-system. The `ABsolver` uses a non-linear optimization tool for Boolean and polynomial arithmetic and a lazy SMT procedure to perform a faster satisfiability checking. Similarly, `CalCs` [32] is also an SMT solver that combines convex optimization and lazy SMT to determine the satisfiability of conjunctions of convex non-linear constraints. Recently, Shoukry *et al.* [33] show that a particular class of logic formulas (named SMC formulas) generalizes a wide range of formulas over Boolean and nonlinear real arithmetic, and propose the Satisfiability Modulo Convex Optimization to solve satisfiability problems over SMC formulas. Our work differs from those previous studies [31–33] since it does not focus on speeding up SMT solvers, but it employs an SMT-based model-checking tool to guide (via counterexample) an optimization search procedure in order to ensure the global optimization.

Recently, νZ [34] extends the SMT solver Z3 for linear optimization problems; Li *et al.* proposed the `SYMBA` algorithm [35], which is an SMT-based symbolic optimization algorithm that uses the theory of linear real arithmetic and SMT solver as black box. Sebastiani and Trentin [36] present `OptiMathSat`, which is 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; in Sebastiani and Tomasi [37], the authors used a combination

of SMT and LP techniques to minimize rational functions; the related work [38] extends their work with linear arithmetic on the mixed integer/rational domain, thus combining SMT, LP, and ILP techniques.

As an application example, Pavlinovic *et al.* [39] propose an approach which considers all possible compiler error sources for statically typed functional programming languages and reports the most useful one subject to some usefulness criterion. The authors formulate this approach as an optimization problem related to SMT and use νZ to compute an optimal error source in a given ill-typed program. The approach described by Pavlinovic *et al.*, which uses MaxSMT solver νZ, shows a significant performance improvement if compared to previous SMT encodings and localization algorithms.

Most previous studies related to SMT-based optimization can only solve linear problems over integer, rational, and Boolean domains in specific cases, leading to limitations in practical engineering applications. Only a few studies [30] are able to solve non-linear problems, but they are also constrained to convex functions. In contrast, this paper proposes a novel counterexample guided inductive optimization method based on SMT solvers to minimize functions, linear or non-linear, convex or non-convex, continuous or discontinuous. As a result, the proposed methods are able to solve optimization problems directly on the rational domain with adjustable precision, without using any other technique to assist the state-space search. Furthermore, our proposed methods employ a model-checking tool to generate automatically SMT formulas from an ANSI-C model of the optimization problem, which makes the representation of problems for SMT solving easy for engineers.

3. Preliminaries

3.1. Optimization Problems Overview

Let $f : X \to \mathbb{R}$ be a cost function, such that $X \subset \mathbb{R}^n$ represents the decision variables vector $x_1, x_2, ..., x_n$ and $f(x_1, x_2, ..., x_n) \equiv f(\mathbf{x})$. Let $\Omega \subset X$ be a subset settled by a set of constraints.

Definition 1. *A multi-variable optimization problem consists in finding an optimal vector $\boldsymbol{x}$, which minimizes f in Ω.*

According to Definition 1, an optimization problem can be written as

$$\begin{array}{ll} \min & f(\mathbf{x}), \\ \text{s.t.} & \mathbf{x} \in \Omega. \end{array} \tag{1}$$

In particular, this optimization problem can be classified in different ways w.r.t. constraints, decision variables domain, and nature of cost function f. All optimization problems considered here are constrained, *i.e.*, decision variables are constrained by the subset Ω. The optimization problem domain X that contains Ω can be the set of $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$. Depending on the domain and constraints, the optimization search-space can be small or large, which influences the optimization algorithms performance.

The cost function can be classified as linear or non-linear; continuous, discontinuous or discrete; convex or non-convex. Depending on the cost

function nature, the optimization problem can be hard to solve, given the time and memory resources [40]. Particularly, non-convex optimization problems are the most difficult ones w.r.t. the cost function nature. A non-convex cost function is a function whose epigraph is a non-convex set and consequently presents various inflexion points that can trap the optimization algorithm to a sub-optimal solution. A non-convex problem is necessarily a non-linear problem and it can also be discontinuous. Depending on that classification, some optimization techniques are unable to solve the optimization problem, and some algorithms usually point to suboptimal solutions, *i.e.*, a solution that is not a global minimum of f, but it only locally minimizes f. Global optimal solutions of the function f, aforementioned, can be defined as

Definition 2. *A vector $\boldsymbol{x}^* \in \Omega$ is a global optimal solution of f in Ω iff $f(\boldsymbol{x}^*) \leq f(x), \forall \boldsymbol{x} \in \Omega$.*

Global optimization is an old problem that remained marginal for a long time period due to difficulties to deal with. Recently, several studies related to global optimization came up, but the proposed solutions are very specific for a particular class of problems [41]. An example of particular solution consists in using differential calculus principles to optimize convex function. Several non-convex functions are globally optimized by transforming the non-convex problem into a set of convex problems [42]. A review about global optimization techniques is provided in the related literature [43, 44].

3.2. Optimization Techniques

Different optimization problems offer different difficulties to their particular solutions. Such complexity is mainly related to the ruggedness (*e.g.*, continuity, differentiability, and smoothness) and dimensionality of the problem (*i.e.*, the dimension, and for the finite case, the number of elements of Ω). Depending on these factors, different optimization techniques can be more efficient to solve a particular optimization problem. Generally, traditional optimization techniques can be divided into two groups: deterministic and stochastic optimization.

The deterministic techniques employ a search engine, where each step is directly and deterministically related to the previous steps [45]. In summary, deterministic techniques can be gradient-based or enumerative search-based. Gradient-based techniques search for points, where the gradient of cost function is null ($\nabla f(\mathbf{x}) = 0$), *e.g.*, gradient-descent [46] and Newton's optimization [1]. Although they are fast and efficient, those techniques are unable to solve non-convex or non-differentiable problems. Enumerative search-based optimization consists in scanning the search-space by enumerating all possible points and comparing cost function with best previous values, *e.g.*, dynamic programming, branch and bound [47], and pattern search [48].

Stochastic techniques employ randomness to avoid the local minima and to ensure the global optimization; such techniques are usually based on metaheuristics [49]. This class of techniques has become very popular in the last decades and has been used in all types of optimization problems. Among those stochastic techniques, simulated annealing [20], particle swarm [18], and

evolutionary algorithms (*e.g.*, genetic algorithms [9]) are usually employed in practice.

Recently, optimization techniques and tools that employ SMT solvers and non-deterministic variables were applied to solve optimization problems [30–32, 34–38, 50], which searches for the global optima in a search-space that is symbolically defined and uses counterexamples produced by SMT solvers to further constrain the search-space. The global optima is the set of values for the decision variables that makes an optimization proposition satisfiable. The technique presented here is the first optimization method based on SMT solvers and inductive generalization described in the literature, which is able to solve non-convex problems over $\mathbb{R}$.

3.3. Model Checking

Model checking is an automated verification procedure to exhaustively check all (reachable) system's states [51]. The model checking procedure typically consists of three steps: modeling, specification, and verification.

Modeling is the first step, where it converts the system to a formalism that is accepted by a verifier. The modeling step usually requires the use of an abstraction to eliminate irrelevant (or less) important system details [52]. The second step is the specification, which describes the system's behavior and the property to be checked. An important issue in the specification is the correctness. Model checking provides ways to check whether a given specification satisfies a system's property, but it is difficult to determine whether such specification covers all properties in which the system should satisfy.

Finally, the verification step checks whether a given property is satisfied w.r.t. a given model, *i.e.*, all relevant system states are checked to search for any state that violates the verified property. In case of a property violation, the verifier reports the system's execution trace (counterexample), which contains all steps from the (initial) state to the (bad) state that lead to the property violation. Errors could also occur due to incorrect system modeling or inadequate specification, thus generating false verification results.

3.3.1. Bounded Model Checking (BMC)

BMC is an important verification technique, which has presented attractive results over the last years [53]. BMC techniques based on Boolean Satisfiability (SAT) [54] or Satisfiability Modulo Theories (SMT) [55] have been successfully applied to verify single- and multi-threaded programs, and also to find subtle bugs in real programs [56, 57]. BMC checks the satisfiability of the negation of a given property at a given depth over a transition system M [54].

Definition 3. *[54] – A set of formulas $\{p_1, p_2, ..., p_n\}$ is said to be satisfiable if there is some structure $\mathcal{A}$ in which all its component formulas are true, i.e., $\{p_1, p_2, ..., p_n\}$ is* `SAT` *iff $\mathcal{A} \models p_1 \wedge \mathcal{A} \models p_2 ... \wedge \mathcal{A} \models p_n$.*

Definition 4. *[54] – Given a transition system M, a property ϕ, and a bound k; BMC unrolls the system k times and translates it into a verification condition (VC) ψ, which is satisfiable iff ϕ has a counterexample of depth less than or equal to k.*

In this study, the ESBMC tool [58] is used as verification engine, as it represents one of the most efficient BMC tools that participated in the last software verification competitions [53]. ESBMC finds property violations such as pointer safety, array bounds, atomicity, overflows, deadlocks, data race, and memory leaks in single- and multi-threaded C/C++ software. It also verifies programs that make use of bit-level, pointers, structures, unions, fixed- and floating-point arithmetic. Inside ESBMC, the associated problem is formulated by constructing the following logical formula

$$\psi_k = I(S_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} (\gamma(s_j, s_{j+1}) \wedge \overline{\phi(s_i)}) \tag{2}$$

where ϕ is a property and S_0 is a set of initial states of M, and $\gamma(s_j, s_{j+1})$ is the transition relation of M between time steps j and $j+1$. Hence, $I(S_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents the executions of a transition system M of length i. The above VC ψ can be satisfied if and only if, for some $i \leq k$ there exists a reachable state at time step i in which ϕ is violated. If the logical formula (2) is satisfiable (*i.e.*, returns *true*), then the SMT solver provides a satisfying assignment (counterexample).

Definition 5. *A counterexample for a property ϕ is a sequence $s_0, s_1, \ldots, s_i$ with $s_0 \in S_0$, $s_i \in S_i$, and $\gamma(s_i, s_{i+1})$ for $i \leq k$ that makes (2) satisfiable. If it is unsatisfiable (i.e., returns false), then one can conclude that there is no error state in k steps or less.*

In addition to software verification, ESBMC has been applied to ensure correctness of digital filters and controllers [59–61]. Recently, ESBMC has been applied to optimize HW/SW co-design [26–28].

3.4. Error and Accuracy

Numbers stored in computers do not have infinite precision; instead, they are represented by a fixed number of bits, *i.e.*, binary digits. Computers allow the programmer to choose different representations (or data types). In particular, data types cannot only differ in the number of bits used, but also in the more fundamental respect of whether the stored number is represented in fixed- or floating-point format (*e.g.*, `float` or `double` types).

3.4.1. Numerical Representation

Aforementioned, computers use basically two different ways of representing a number, fixed- or floating-point. In a fixed-point representation, a number is represented by three fixed parts: sign, integer part, and fractional part. For instance, to store a number using a 32-bit format, 1 bit is reserved for the sign, 15 bits for the integer part and 16 bits for the fractional part. Note that for this case, $2^{-16} \approx 0.00001526$ is the gap between two adjacent fixed-point numbers. Thus, a number whose representation exceeds 32 bits would be stored inexactly.

A common problem in fixed-point representation is overflow, which occurs when at some stage during processing binary arithmetic, a number outside the finite range of representation is generated.

Alternatively, there is the floating-point representation, where a number is represented internally by a sign S (interpreted as plus or minus), an exact integer exponent E, and an exactly representation binary mantissa M as

$$S \times M \times b^{E-e}, \tag{3}$$

where b is the representation base, which is usually $b = 2$ since multiplication and division operations by 2 can be accomplished by left- or right-shift of the bits; and e is the exponent bias, an fixed integer constant fixed for any machine and representation, *e.g.*, for 32-bit `float` values, the exponent is represented in 8 bits ($e = 127$) and for 64-bit `double` values, the exponent is represented in 11 bits ($e = 1023$).

For 32-bit representation, the smallest normalized positive number is $2^{-126} \approx 1.18 \times 10^{-38}$, which is much less than in a fixed-point representation. Futhermore, the spacing between the floating-point numbers is not uniform, as we move away from the origin, the spacing becomes less dense.

Most modern processors adopt the same floating-point data representation, as specified by the IEEE standard 754-1985 [62]. The floating-point seems more appropriate when you need a certain precision for the value. However, when absolute precision is required, fixed-point represents a more appropriate choice.

3.4.2. Roundoff and Truncation Errors

In general, the floating-point representation behaves very similar to real numbers. However, there are many inconsistencies between the behavior of floating-point numbers in base 2 and real numbers. The main causes of error in the calculation of floating-point are: *roundoff* and *truncation errors*.

Arithmetic among numbers in floating-point representation is not exact, even if the operands happen to be exactly represented, *i.e.*, they have exact values in the form of Eq. (3). Machine accuracy, ϵ_m, is the smallest (in magnitude) floating-point number, which must be added to the floating-point number 1.0 to produce a floating-point result different from 1.0. IEEE 754 standard `float` has ϵ_m about 1.19×10^{-7}, while `double` has about 2.22×10^{-16}. The machine accuracy is the fractional accuracy which floating-point numbers are represented, corresponding to a change of one in the least significant bit of the mantissa. Almost any arithmetic operation among floating-point numbers should be considered as introducing an additional fractional error of at least ϵ_m. This type of error is called *roundoff* error.

Roundoff error is a characteristic of computer hardware. There is another kind of error that is a characteristic of the program or algorithm used, independent on the hardware on which the program is executed. Many numerical algorithms compute discrete approximations to some desired continuous quantity. In these cases, there is an adjustable parameter; any practical calculation is done with a finite, but sufficiently large, choice of that parameter. The discrepancy between the true answer and the answer obtained in a practical calculation is called the *truncation* error. Truncation error would persist even on a perfect computer that had an infinitely accurate representation and no roundoff error.

As a general rule, there is not much that a programmer can do about roundoff error. However, truncation error is entirely under the control of the programmer.

4. Counterexample Guided Inductive Optimization of Non-convex Function

4.1. Modeling Optimization Problems using a Software Model Checker

There are two important directives in the C/C++ programming language, which can be used for modeling and controlling a verification process: `ASSUME` and `ASSERT`. The `ASSUME` directive can define constraints over (non-deterministic) variables, and the `ASSERT` directive is used to check system's correctness w.r.t. a given property. Using these two statements, any off-the-shelf C/C++ model checker (*e.g.*, CBMC [57], CPAChecker [63], and ESBMC [58]) can be applied to check specific constraints in optimization problems, as described by Eq. (1).

Here, the verification process is iteratively repeated to solve an optimization problem using intrinsic functions available in ESBMC (*e.g.*, *__ESBMC_assume* and *__ESBMC_assert*). We apply incremental BMC to efficiently prune the state-space search based on counterexamples produced by an SMT solver. Note that completeness is not an issue here (cf. Definitions 1 and 2) since our optimization problems are represented by loop-free programs [64].

If the minimum candidate value of a function defined by Eq. (1) is given (f_c), then the ASSERT directive can be used to check the $l_{optimal}$ satisfiability as

$$l_{optimal} \iff f(\mathbf{x}) \geq f_c. \tag{4}$$

Note that if $\neg l_{optimal}$ is unsatisfiable, then f_c is the function minimum; otherwise, there is some state-space portion, defined by the problem constraints, which violates the property.

Although, this seems simple, some problems arise due to finite-precision arithmetic implemented by the verifiers. For instance, in ESBMC, the SMT solvers Z3 and MathSAT support floating-point arithmetic, but Boolector only supports fixed-point arithmetic. Thus, error and accuracy issues previously described can occur. In order to avoid that, the literal $l_{optimal}$ is modified for $l_{suboptimal}$ given by Eq. (5) as

$$l_{suboptimal} \iff f(\mathbf{x}) > f_c - \delta, \tag{5}$$

where δ must be sufficiently high to reduce the effects of roundoff and truncation errors in the computations. By contrast, if $\neg l_{suboptimal}$ is unsatisfiable, then f_c is not the function minimum, but it will be at one distance limited by δ of the minimum value. As described in the following sections, δ can also be used with the purpose of determining the minimum improvement at each iteration in the cost function. The basic idea is to use the ability of verifiers to check the satisfiability of a given property and then return a counterexample that contains the error trace. Thus, through successive satisfiability checks of the literal $\neg l_{suboptimal}$, we can guide the verification process to solve the optimization problem given by Eq. (1).

4.2. CEGIO: the Generalized Algorithm (CEGIO-G)

The generalized SMT-based optimization algorithm previously presented by Araújo *et al.* [13] is able to find the global optima for any optimization problem, given a precision for the respective decision variables. The execution time of that algorithm depends on how the state-space search is restricted and on the number of the solution decimal places. Specifically, the algorithm presents a fixed-point solution with adjustable precision, *i.e.*, the number of decimal places can be defined.

However, this algorithm might take long for achieving the optimal solution of unconstrained optimization problems with non-integer solutions, since it depends on the aforementioned precision. Although this algorithm usually produces a longer execution time than other traditional techniques, its error rate is typically lower than other existing methods, once it is based on a complete and sound verification procedure.

Alg. 1 shows an improved version of the algorithm presented by Araújo *et al.* [13]; this algorithm is denoted here as "Generalized CEGIO algorithm" (CEGIO-G). Note that Alg. 1 contains two nested loops, the outer (`for`) loop is related to the desired precision and the inner (`while`) loop is related to the verification process. This configuration speeds up the optimization problem due to the complexity reduction if compared to the algorithm originally described by Araújo *et al.* [13]. The CEGIO-G algorithm uses the manipulation of fixed-point number precision to ensure the optimization convergence.

Alg. 1 has four inputs: a cost function $f(x)$; the space for constraint set Ω; a number of decimal places of decision variables η; and the minimum improvement value for the candidate cost function δ; it also provides two outputs: the decision variables vector $\mathbf{x}^*$ and the cost function minimum value $f(\mathbf{x}^*)$. After the variable initialization and declaration (lines 1-2 of Alg. 1), the search domain Ω^ϵ is specified in line 4, which is defined by lower and upper bounds of the auxiliary variables $\mathbf{X}$. Note that they are declared as non-deterministic integer variables; otherwise, if declared as non-deterministic float variables, then the state-space search would considerably increase. Thus, $Omega^\epsilon$ limits are given by Eq. (6).

$$lim\{\Omega^\epsilon\} = lim\{\Omega\} \times 10^\epsilon \tag{6}$$

The variable ϵ is used to handle the auxiliary variables and to obtain the decision variables value, $\mathbf{x}$, such that ϵ defines the number of decimal places of $\mathbf{x}$. An null value of ϵ results in integer solutions. Solutions with one decimal place is obtained for $\epsilon = 1$, two decimal places are achieved for $\epsilon = 2$.

Initially, ϵ is equal to zero and must be updated at the end of the each iteration of the outer (`for`) loop, such that it will increase the decision variables domain by one decimal place in the next iteration of the loop. After the problem constraint definition, the model for function $f(x)$ is defined (in line 5), considering the decision variables decimal places, *i.e.*, $\mathbf{x} = \mathbf{X}/10^\epsilon$.

At each iteration of the inner (`while`) loop, the satisfiability of $\neg l_{suboptimal}$ given by Eq. (5) is checked. Before that, however, a new constraint is placed on the state-space by taking into account the cost function value for each region of the state-space (see line 8), *i.e.*, there is no need for verifying values

input : A cost function $f(\mathbf{x})$, the space for constraint set Ω, a number of decimal places of decision variables η, and the minimum improvement value of the function δ

output: The optimal decision variable vector $\mathbf{x}^*$, and the optimal function value $f(\mathbf{x}^*)$

Initialize $f_c^{(0)}$ *randomly and* $i = 0$
Declare the auxiliary variables $\boldsymbol{X}$ *as non-deterministic integer variables*
for $\epsilon = 0 \rightarrow \eta;\ \epsilon \in \mathbb{Z}$ **do**
    Define bounds for $\boldsymbol{X}$ *with the* `ASSUME` *directive, such that* $\boldsymbol{X} \in \Omega^{\epsilon}$
    Describe a model for objective function $f(\boldsymbol{x})$*, where* $\boldsymbol{x} = \boldsymbol{X}/10^{\epsilon}$
    Do the auxiliary variable Check = TRUE
    while *Check* **do**
        Constrain $f(\boldsymbol{x}^{(i)}) < f_c^{(i)}$ *with the* `ASSUME` *directive*
        Verify the satisfiability of $\neg l_{suboptimal}$ *given by Eq.* (5) *with the* `ASSERT` *directive*
        if $\neg l_{suboptimal}$ *is satisfiable* **then**
            Update $\boldsymbol{x}^* = \boldsymbol{x}^{(i)}$ *and* $f(\boldsymbol{x}^*) = f(\boldsymbol{x}^{(i)})$ *based on the counterexample*
            Do $i = i + 1$
            Do $f_c^{(i)} = f(\boldsymbol{x}^*) - \delta$
        end
        else
            Check = `FALSE`
        end
    end
end
return $\boldsymbol{x}^*$ *and* $f(\boldsymbol{x}^*)$

Algorithm 1: CEGIO-G: the generalized algorithm.

greater than the minimum candidate value, since the algorithm searches for the minimum value of the cost function. In this specific step, the search-space is remodelled for the i-th precision and it employs previous results of the optimization process.

The verification step is performed in lines 9-10, where the candidate function $f_c^{(i)}$, *i.e.*, $f(\mathbf{x}^*) - \delta$ if $i > 0$, is analyzed by means of the satisfiability check of $\neg l_{suboptimal}$. If there is a $f(\mathbf{x}) \leq f_c^{(i)}$ that violates the `ASSERT` directive, then the decision variables vector and minimum value of cost function are updated based on the counterexample, futhermore, the candidate function is updated, $f_c^{(i)} = f(\mathbf{x}^*) - \delta$ (line 13), after the increase of i, and the algorithm returns to remodel the state-space again (line 8). Note that, *delta* define the least improvement in the cost function w.r.t the previous iteration.

If the `ASSERT` directive is not violated, the last candidate f_c is the minimum value with the precision variable ϵ; thus, ϵ incremented from one in the `for` loop to η, adding a decimal place to the optimization solution, and the outer (`for`) loop is repeated. The algoritm concludes if the outer loop achieves the limit defined by the input parameter η; thus it returns the optimal vector of decision variables with η decimal places and the optimal value of the cost function.

4.3. Proof of Convergence

A generic optimization problem described in the previous section is formalized as follow: given a set $\Omega \subset \mathbb{R}^n$, determine $\mathbf{x}^* \in \Omega$, such that, $f(\mathbf{x}^*) \in \Phi$ is the lowest value of the function f, *i.e.*, $\min f(\mathbf{x})$, where $\Phi \subset \mathbb{R}$ is the image set of f (*i.e.*, $\Phi = Im(f)$). Our approach solves the optimization problem with η decimal places, *i.e.*, the solution $\mathbf{x}^*$ is an element of the rational domain $\Omega^\eta \subset \Omega$ such that $\Omega^\eta = \Omega \cap \Theta$, where $\Theta = \{\mathbf{x} \in \mathbb{Q} | \mathbf{x} = k \times 10^{-\eta}, \forall k \in \mathbb{Z}\}$, *i.e.*, Ω^η is composed by rationals with η decimal places in Ω (*e.g.*, $\Omega^0 \subset \mathbb{Z}$). Thus, $\mathbf{x}^{*,\eta}$ is the minimum of function f in Ω^η with η decimal places.

Lemma 1. *Let Φ be a finite set composed by all values $f(\boldsymbol{x}) < f_c^{(i)}$, where $f_c^{(i)} \in \Phi$ is any minimum candidate and $\boldsymbol{x} \in \Omega$. The literal $\neg l_{suboptimal}$ (Eq. 5) is UNSAT iff $f_c^{(i)}$ holds the lowest values in Φ; otherwise, $\neg l_{suboptimal}$ is SAT iff there exists any $\boldsymbol{x}^{(i)} \in \Omega$ such that $f(\boldsymbol{x}^{(i)}) < f_c^{(i)}$.*

Theorem 1. *Let Φ_i be the i-th image set of the optimization problem constrained by $\Phi_i = \{f(\boldsymbol{x}) < f_c^{(i)}\}$, where $f_c^{(i)} = f(\boldsymbol{x}^{*,(i-1)}) - \delta, \forall i > 0$, and $\Phi_0 = \Phi$. There exists an $i^* > 0$, such that $\Phi_{i^*} = \emptyset$, and $f(\boldsymbol{x}^*) = f_c^{(i^*)}$.*

Proof. Initially, the minimum candidate $f_c^{(0)}$ is chosen randomly from Φ_0. Considering Lemma 1, if $\neg l_{optimal}$ is SAT, any $f(\mathbf{x}^{*,(0)})$ (from the counterexample) is adopted as next candidate solution, *i.e.*, $f_c^{(1)} = f(\mathbf{x}^{*,(0)}) - \delta$, and every element from Φ_1 is less than $f_c^{(1)}$. Similarly in the next iterations, while $\neg l_{optimal}$ is SAT, $f_c^{(i)} = f(\mathbf{x}^{*,(i-1)}) - \delta$, and every element from Φ_i is less than $f_c^{(i)}$, consequently, the number of elements of Φ_{i-1} is always less than that of Φ_i. Since Φ_0 is finite,

in the i^*-th iteration, Φ_{i^*} will be empty and the $\neg l_{optimal}$ is UNSAT, which leads to (Lemma 1) $f(\mathbf{x}^*) = f_c^{(i^*)}$. □

Theorem 1 provides sufficient conditions for the global minimization over a finite set; it solves the optimization problem defined at the beginning of this section, *iff* the search domain Ω^η is finite. It is indeed finite, once it is defined as an intersection between a bounded set (Ω) and a discrete set (Θ). Thus, the CEGIO-G algorithm will always provide the minimum $\mathbf{x}^*$ with η decimal places.

4.4. Illustrative Example

The Ursem03's function is employed to illustrate the present SMT-based optimization method for non-convex optimization problems [17]. The Ursem03's function is represented by a two-variables function with only one global minimum in $f(x_1, x_2) = -3$, and has four regularly spaced local minima positioned in a circumference, with the global minimum in the center. Ursem03's function is defined by Eq. (7); Fig. 1 shows its respective graphic.

$$f(x_1, x_2) = -\sin\left(2.2\pi x_1 - \frac{\pi}{2}\right)\frac{(2-|x_1|)(3-|x_1|)}{4} - \sin\left(2.2\pi x_2 - \frac{\pi}{2}\right)\frac{(2-|x_2|)(3-|x_2|)}{4} \quad (7)$$

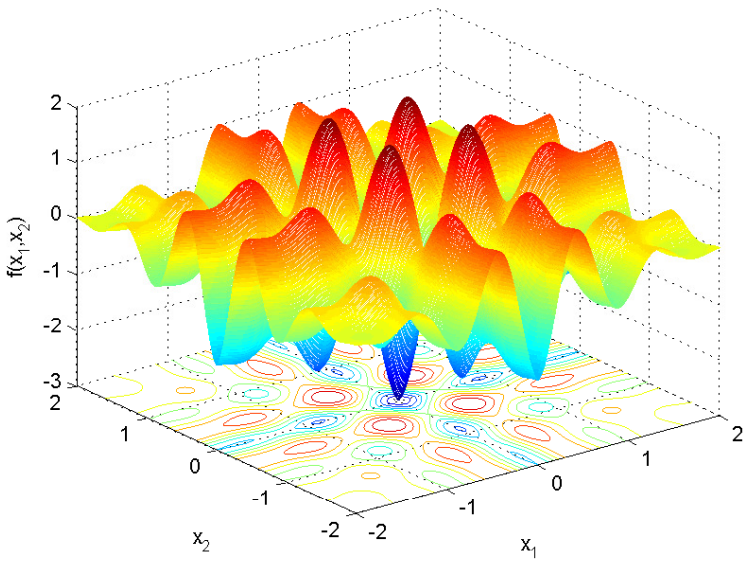

Figure 1: *Ursem03's function*

4.4.1. Modeling

The modeling process defines constraints, *i.e.*, Ω boundaries (cf. Section 3.1). This step is important for reducing the state-space search and consequently for

avoiding the state-space explosion by the underlying model-checking procedure. Our verification engine is not efficient for unconstrained optimization; fortunately, the verification time can be drastically reduced by means of a suitable constraint choice. Consider the illustrative optimization problem given by Eq. (8), which is related to the Ursem03's function given in Eq. (7), where the constraints are defined by half-closed intervals that lead to a large domain.

$$\begin{array}{ll} \min & f(x_1, x_2) \\ \text{s.t.} & x_1 \geq 0, \\ & x_2 \geq 0. \end{array} \tag{8}$$

Note that inequalities $x_1 \geq 0$ and $x_2 \geq 0$ are pruning the state-space search to the first quadrant; however, even so it produces a (huge) state-space to be explored since x_1 and x_2 can assume values with very high modules. The optimization problem given by Eq. (8) can be properly rewritten as Eq. (9) by introducing new constraints. The boundaries are chosen based on the study described by Jamil and Yang [17], which defines the domain in which the optimization algorithms can evaluate the benchmark functions, including the Ursem03's function.

$$\begin{array}{ll} \min & f(x_1, x_2) \\ \text{s.t.} & -2 \leq x_1 \leq 2, \\ & -2 \leq x_2 \leq 2. \end{array} \tag{9}$$

From the optimization problem definition given by Eq. (9), the modeling step can be encoded, where decision variables are declared as non-deterministic variables constrained by the `ASSUME` directive. In this case, $-2 \leq x_1 \leq 2$ and $-2 \leq x_2 \leq 2$. Fig. 2 shows the respective C code for modeling Eq. (9). Note

```
#include "math2.h"
float nondet_float();
int main() {
  //define decision variables
  float x1 = nondet_float();
  float x2 = nondet_float();
  //constrain the state-space search
  __ESBMC_assume((x1>=-2) && (x1<=2));
  __ESBMC_assume((x2>=-2) && (x2<=2));
  //computing Ursem's function
  float fobj;
  fobj= -sin(2.2*pi*x1-pi/2)*(2-abs(x1))(3-abs(x1))/4
  -sin(2.2*pi*x2-pi/2)*(2-abs(x2))(3-abs(x2))/4;
  return 0;
}
```

Figure 2: *C Code for the optimization problem given by Eq.* (9).

that in Figure 2, the decision variables x_1 and x_2 are declared as floating-point numbers initialized with non-deterministic values; we then constraint the state-space search using assume statements. The objective function of Ursem's function is then declared as described by Eq. 7.

4.4.2. Specification

The next step of the proposed methodology is the specification, where the system behavior and the property to be checked are described. For the Ursem03's function, the result of the specification step is the C program shown in Fig. 3, which is iteratively checked by the underlying verifier.

Note that the decision variables are declared as integer type and their initialization depends on a given precision p, which is iteratively adjusted once the counterexample is produced by the SMT solver. Indeed, the C program shown in Fig. 2 leads the verifier to produce a considerably large state-space exploration, if the decision variables are declared as non-deterministic floating-point type. In this study, decision variables are defined as non-deterministic integers, thus discretizing and reducing the state-space exploration; however, this also reduces the optimization process precision. In particular, the accuracy requirement depends on the application and problem domain. A high accuracy can be desirable in some particular applications. However, the proposed optimization algorithms are able to optimize target functions for any desired accuracy defined by the user, given the imposed time and memory limits.

To trade-off both precision and verification time, and also to maintain convergence to an optimal solution, the underlying model-checking procedure has to be iteratively invoked, in order to increase its precision for each successive execution, as follows

$$p = 10^{\epsilon}. \tag{10}$$

An integer variable p is created and iteratively adjusted, such that ϵ represents the amount of decimal places related to the decision variables, *i.e.*, $0 \leq \epsilon \leq \eta$, as discussed in Section 4.1. Additionally, a new constraint is inserted; in particular, the new value of the objective function $f(\mathbf{x}^{(i)})$ at the i-th must not be greater than the value obtained in the previous iteration $f(\mathbf{x}^{*,(i-1)})$. Initially, all elements in the state-space search Ω are candidates for optimal points, and this constraint cutoffs several candidates on each iteration.

In addition, a property has to be specified to ensure convergence to the minimum point on each iteration. This property specification is stated by means of an assertion, which checks whether the literal $\neg l_{suboptimal}$ given in Eq. (5) is satisfiable for every value, $f(\mathbf{x})$, remaining in the state-space search (*i.e.*, traversed from lowest to highest).

The verification procedure stops when the literal $\neg l_{suboptimal}$ is unsatisfiable, *i.e.*, if there is any $\mathbf{x}^{(i)}$ for which $f(\mathbf{x}^{(i)}) \leq f_c$; a counterexample shows such $\mathbf{x}^{(i)}$, converging iteratively $f(\mathbf{x})$ from the optimal $f(\mathbf{x}^*)$. Fig. 3 shows the initial specification for the optimization problem given by Eq. (9). The initial candidate value of the objective function can be randomly initialized. For the example shown in Fig. 3, $f_c^{(0)}$ is arbitrarily initialized to 100, but the present optimization algorithm works for any initial state, and for $i > 0$, $f_c^{(i+1)} = f(\mathbf{x}^{*,(i)}) - \delta$, as specified in line 13 of the Alg. 1.

Note that, the code illustrated in Fig. 3 must be executed iteratively with the aforementioned f_c and p variables updated; this can be done by iteratively rewriting that code by means of scripts.

```
#include "math2.h"
#define p 1 //precision variable
int nondet_int();
float nondet_float();
int main() {
  float f_c = 100;  //candidate value of objective function
  int lim_inf_x1 = -2*p;
  int lim_sup_x1 = 2*p;
  int lim_inf_x2 = -2*p;
  int lim_sup_x2 = 2*p;
  int X1 = nondet_int();
  int X2 = nondet_int();
  float x1 = float nondet_float();
  float x2 = float nondet_float();
  __ESBMC_assume( (X1>=lim_inf_x1) && (X1<=lim_sup_x1) );
  __ESBMC_assume( (X2>=lim_inf_x2) && (X2<=lim_sup_x2) );
  __ESBMC_assume( x1 = (float) X1/p );
  __ESBMC_assume( x2 = (float) X2/p );
  float fobj;
  fobj= -sin2(2.2*pi*x1-pi/2)*(2-abs2(x1))(3-abs2(x1))/4
  -sin2(2.2*pi*x2-pi/2)*(2-abs2(x2))(3-abs2(x2))/4;
  //constrain to exclude fobj>f_c
  __ESBMC_assume( fobj < f_c );
  assert( fobj > f_c );
  return 0;
}
```

Figure 3: *C code after the specification of Eq.* (9).

4.4.3. Verification

Finally, in the verification step, the C program shown in Fig. 3 is checked by the verifier and a counterexample is returned with a set of decision variables $\mathbf{x}$, for which the objective function value converges to the optimal value. A specified C program only returns a successful verification result if the previous function value is the optimal point for that specific precision (defined by p), *i.e.*, $f_c^{(i+1)} = f(\mathbf{x}^{*,(i)}) - \delta$. For the example shown in Fig. 3, the verifier shows a counterexample with the following decision variables: $x_1 = 2$ and $x_2 = 0$. These decision variable are used to compute a new minimum candidate, note that $f(2,0) = -1.5$, which is the new minimum candidate solution provided by this verification step. Naturally, it is less than the initial value (100), and this verification can be repeated with the new value of $f_c^{(i+1)}$, in order to obtain an objective function value that is close to the optimal point on each iteration. Note that the data provided by the counterexample is crucial for the algorithm convergence and for the state-space search reduction.

4.4.4. Avoiding the Local Minima

As previously mentioned, an important feature of this proposed CEGIO method is always to find the global minimum with precision of η decimal

places (cf. Theorem 1). Many optimization algorithms might be trapped by local minima and they might incorrectly solve optimization problems. However, the present technique ensures the avoidance of those local minima, through the satisfiability checking, which is performed by successive SMT queries. This property is maintained for any class of functions and for any initial state.

Figures 4 and 5 show the aforementioned property of this algorithm, comparing its performance to the genetic algorithm (GA). In those figures, Ursem03's function is adapted for a single-variable problem over x_1, *i.e.*, x_2 is considered fixed and equals to 0.0, and the respective function is reduced to a plane crossing the global optimum in $x_1 = 0$. The partial results after each iteration are illustrated by the various marks in these graphs; for GA, we show only the best result of each algorithm epoch. Note that the present method does not present continuous trajectory from the initial point to the optimal point; however, it always achieves the correct solution. Fig. 4 shows that both techniques (GA and SMT) achieve the global optimum. However, Fig. 5 shows that GA might be trapped by the local minimum for a different initial point. In contrast, the proposed CEGIO method can be initialized further away from the global minimum and as a result it can find the global minimum after some iterations, as shown in Figures 4 and 5.

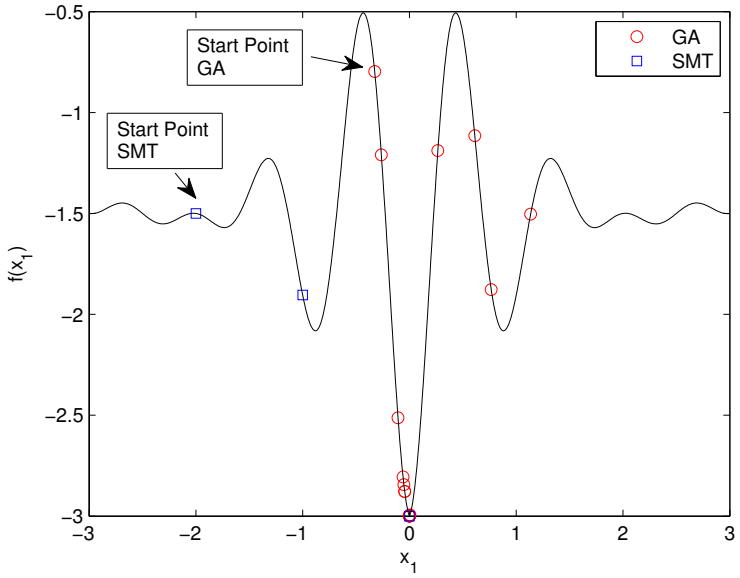

Figure 4: *Optimization trajectory of GA and SMT for a Ursem03's plane in $x_2 = 0$. Both methods obtain the correct answer.*

5. Counterexample Guided Inductive Optimization of Special Functions

This section presents two variants of the Counterexample Guided Inductive Optimization (CEGIO) algorithm for global constrained optimization. A simpli-

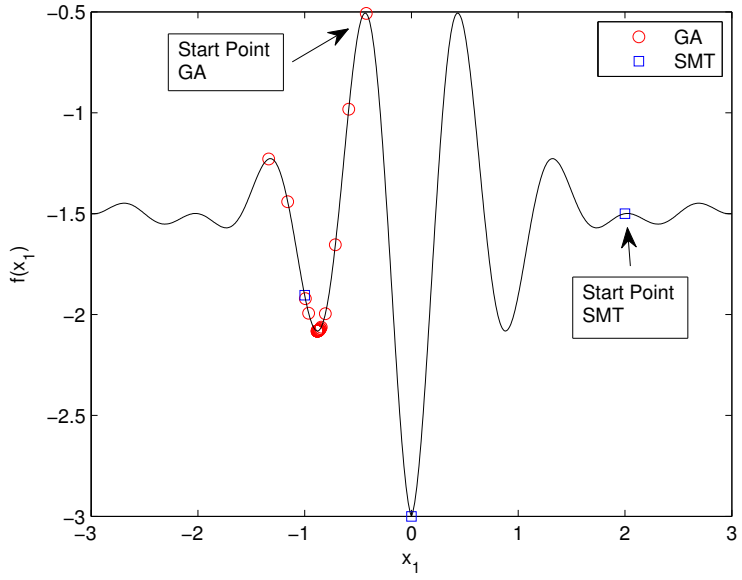

Figure 5: *Optimization trajectory of GA and SMT for a Ursem03's plane in $x_2 = 0$. GA is trapped by an local minimum, but SMT obtains the correct answer.*

fied CEGIO algorithm (CEGIO-S) is explained in Subsection 5.1, while Subsection 5.2 presents the fast CEGIO algorithm (CEGIO-F) for convex optimization problems. Additionally, a convergence proof of the CEGIO-F algorithm is described in Subsection 5.2.2.

5.1. A Simplified Algorithm for CEGIO (CEGIO-S)

Alg. 1 is suitable for any class of functions, but there are some particular functions that contain further knowledge about their behaviour (e.g., positive semi-definite functions such as $f(\mathbf{x}) \geq 0$. Using that knowledge, Alg. 1 is slightly modified for handling this particular class of functions. This algorithm is named here as "Simplified CEGIO algorithm" (CEGIO-S) and it is presented in Alg. 2.

Note that Alg. 2 contains three nested loops after the variable initialization and declaration (lines 1-3), which is similar to the algorithm presented in [13]. In each execution of the outer loop (`for`) (lines 4-29), the bounds and precision are updated accordingly. The main difference in this algorithm w.r.t the Alg. 1 is the presence of the condition in line 8, *i.e.*, it is not necessary to generate new checks if that condition does not hold, since the solution is already at the minimum limit, *i.e.*, $f(\mathbf{x}^*) = 0$.

Furthermore, there is another inner (`while`) loop (lines 12-15), which is responsible for generating multiple VCs through the `ASSERT` directive, using the interval between f_m and $f_c^{(i)}$. Note that this loop generates $\alpha + 1$ VCs through the step defined by γ in line 7.

These modifications allow Alg. 2 to converge faster than Alg. 1 for the positive semi-definite or definite functions, since the chance of a check failure is

input : A cost function $f(\mathbf{x})$, the space for constraint set Ω, a number of decimal places of decision variables η, the minimum improvement value of the function δ, and a learning rate α

output: The optimal decision variable vector $\mathbf{x}^*$, and the optimal value of function $f(\mathbf{x}^*)$

Initialize $f_m = 0$
Initialize $f_c^{(0)}$ *randomly and* $i = 0$
Declare the auxiliary variables $\boldsymbol{X}$ *as non-deterministic integer variables*
for $\epsilon = 0 \rightarrow \eta;\ \epsilon \in \mathbb{Z}$ **do**
    Define bounds for $\boldsymbol{X}$ *with the* `ASSUME` *directive, such that* $\boldsymbol{X} \in \Omega^{\epsilon}$
    Describe a model for objective function $f(\boldsymbol{x})$*, where* $\boldsymbol{x} = \boldsymbol{X}/10^{\epsilon}$
    Declare $\gamma = (f(\boldsymbol{x}^{(i-1)}) - f_m)/\alpha$
    if $(f_c^{(i)} - f_m > 10^{-5})$ **then**
        Do the auxiliary variable Check = `TRUE`
        while *Check* **do**
            Constraint $f(\boldsymbol{x}^{(i)}) < f_c^{(i-1)}$ *with the* `ASSUME` *directive*
            while $(f_m \leq f_c^{(i)})$ **do**
                Verify the satisfiability of $l_{suboptimal}$ *given by Eq.* (4) *for each* f_m*, with the* `ASSERT` *directive*
                Do $f_m = f_m + \gamma$
            end
            if $\neg l_{suboptimal}$ *is satisfiable* **then**
                Update $\boldsymbol{x}^* = \boldsymbol{x}^{(i)}$ *and* $f(\boldsymbol{x}^*) = f(\boldsymbol{x}^{(i)})$ *based on the counterexample*
                Do $i = i + 1$
                Do $f_c^{(i)} = f(\boldsymbol{x}^*) - \delta$
            end
            else
                Check = `FALSE`
            end
        end
    end
    else
        break
    end
end
return $\boldsymbol{x}^*$ *and* $f(\boldsymbol{x}^*)$

Algorithm 2: CEGIO-S: a simplified algorithm.

higher due to the larger number of properties. However, if α represents a large number, then the respective algorithm would produce many VCs, which could cause the opposite effect and even lead the verification process to exhaust the memory.

5.2. Counterexample Guided Inductive Optimization of Convex Problems

Convex functions are an important class of functions commonly found in many areas of mathematics, physics, and engineering [65]. A convex optimization problem is similar to Eq. (1), where $f(\mathbf{x})$ is a convex function, which

satisfies Eq. (11) as

$$f(\alpha x_1 + \beta x_2) \leq \alpha f(x_1) + \beta f(x_2) \tag{11}$$

for all $x_i \in \mathbb{R}^n$, with $i = 1, 2$ and all $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1$, $\alpha \geq 0$, $\beta \geq 0$.

Theorem 2 is an important theorem for convex optimization, which is used by most convex optimization algorithms.

Theorem 2. *A local minimum of a convex function f, on a convex subset, is always a global minimum of f [66].*

Here, Theorem 2 is used to ensure convergence of the CEGIO convex optimization algorithm presented in Subsection 5.2.1.

5.2.1. Fast CEGIO (CEGIO-F)

Alg. 1 aforementioned evolves by increasing the precision of the decision variables, *i.e.*, in the first execution of its `for` loop, the obtained global minimum is integer since $\epsilon = 0$, called $\mathbf{x}^{*,0}$. Alg. 3 is an improved algorithm of that Alg. 1 for application in convex functions. It will be denoted here as "Fast CEGIO algorithm" (CEGIO-F).

Note that, the only difference of Alg. 1 is the insertion of line 13, which updates limits of the set Ω^ϵ before of ϵ. For each execution of the `for`-loop, the solution is optimal for precision ϵ. A new search domain $\Omega^\epsilon \subset \Omega^\eta$ is obtained from a CEGIO process over $\Omega^{\epsilon-1}$, for $\epsilon > 0$, defining Ω^ϵ as follows: $\Omega^\epsilon = \Omega^\eta \cap [x^{*,\epsilon-1} - p, x^{*,\epsilon-1} + p]$, where p is given by Eq. (10) e $x^{*,\epsilon-1}$ is the solution with $\epsilon - 1$ decimal places.

5.2.2. Proof of Convergence for the Fast CEGIO Algorithm

The CEGIO-F algorithm computes iteratively for every $\Omega^\epsilon, 0 < \epsilon \leq \eta$. Theorem 1 ensures the global minimization for any finite Ω^ϵ. The global convergence of the CEGIO-F algorithm is ensured *iff* the minima of any $\Omega^{\epsilon-1}$ is inside Ω^ϵ. It holds for the generalized algorithm since $\Omega^1 \subset \Omega^2 ... \subset \Omega^{\epsilon-1} \subset \Omega^\epsilon$. However, the CEGIO-F algorithm modifies Ω^ϵ boundaries using the $\epsilon - 1$-th solution, for $\epsilon > 0$.

Lemma 2. *Let $f : \Omega^\epsilon \to \mathbb{R}$ be a convex function, as Ω^ϵ is a finite set, Theorem 1 ensures that the minimum, $x^{*,\epsilon}$ in Ω^ϵ is a local minimum for precision ϵ, where $\epsilon = \log p$. In addition, as f is a convex function, any element x outside $[x^{*,\epsilon} - p, x^{*,k} + p]$ has its image $f(x) > f(x^{*,\epsilon})$ ensured by Eq.* (11).

Lemma 2 ensures that the solution is a local minimum of f, and Theorem 2 ensures that it is a global minimum. As a result, bounds of Ω^ϵ can be updated on each execution of the outer (`for`) loop; this modification considerably reduces the state-space searched by the verifier, which consequently decreases the algorithm execution time.

6. Experimental Evaluation

This section describes the experiments design, execution, and analysis for the proposed CEGIO algorithms. We use the ESBMC tool as verification

input : A cost function $f(\mathbf{x})$, the space for constraint set Ω, and a desired precision ϵ

output: The optimal decision variable vector $\mathbf{x}^*$, and the optimal value of function $f(\mathbf{x}^*)$

Initialize $f_c^{(0)}$ *randomly and* $i = 0$
Declare the auxiliary variables $\boldsymbol{X}$ *as non-deterministic integer variables*
for $\epsilon = 0 \rightarrow \eta$; $\epsilon \in \mathbb{Z}$ **do**
    Define bounds for $\boldsymbol{X}$ *with the* `ASSUME` *directive, such that* $\boldsymbol{X} \in \Omega^{\epsilon}$
    Describe a model for objective function $f(\boldsymbol{x})$*, where* $\boldsymbol{x} = \boldsymbol{X}/10^{\epsilon}$
    Do the auxiliary variable Check = `TRUE`
    while *Check* **do**
        Constrain $f(\boldsymbol{x}^{(i)}) < f_c^{(i)}$ *with the* `ASSUME` *directive*
        Verify the satisfiability of $\neg l_{suboptimal}$ *given by Eq.* (5) *with the* `ASSERT` *directive*
        if $\neg l_{suboptimal}$ *is satisfiable* **then**
            Update $\boldsymbol{x}^* = \boldsymbol{x}^{(i)}$ *and* $f(\boldsymbol{x}^*) = f(\boldsymbol{x}^{(i)})$ *based on the counterexample*
            Do $i = i + 1$
            Do $f_c^{(i)} = f(\boldsymbol{x}^*) - \delta$
        end
        else
            Check = `FALSE`
        end
    end
    Update limits of the set Ω^{ϵ}
end
return $\boldsymbol{x}^*$ *and* $f(\boldsymbol{x}^*)$

Algorithm 3: CEGIO-F: the fast algorithm.

engine to find the optimal solution for a particular class of functions. We also compare the present approaches to other exisiting techniques, including genetic algorithm, particle swarm, pattern search, simulated annealing, and nonlinear programming. Preliminary results allowed us to improve the experimental evaluation as follows.

(i) There are functions with multiplication operations and large inputs, which lead to overflow in some particular benchmarks. Thus, the data-type `float` is replaced by `double` in some particular functions to avoid overflow.

(ii) We over-approximate floats by fixed-point arithmetic for those solvers that do not support floating-point arithmetic. In particular, this over-approximation might introduce behavior that is not present in a real implementation. However, we still aimed to exploit those SMT solvers (with no support for floating-point arithmetic), in order to check the feasibility of our approach, given that an encoding of the full floating-point arithmetic into the BMC framework typically leads to large formulae to be handled by the SAT solver after bit-blasting.

(iii) ESBMC uses different SMT solvers to perform program verification.

Depending on the selected solver, the results, verification time, and counterexamples can be different. This is observed in several studies [28, 59, 60, 67]; as a result, our evaluation here is also carried out using different SMT solvers such as Boolector [15], Z3 [14], and MathSAT [16], in order to check whether a particular solver heavily influences the performance of the CEGIO algorithms.

(iv) There are functions that present properties which permits the formulation of invariants to prune the state-space search, *e.g.*, functions that use absolute value operators (or polynomial functions with even degree); those functions will always present positive values. As a result, the optimization processes can be simplified, reducing the search domain to positive regions only. Such approach led to the development of Algorithm 2, which aims to reduce the verification time.

All experiments are conducted on a otherwise idle computer equipped with Intel Core i7-4790 CPU 3.60 GHz, with 16 GB of RAM, and Linux OS Ubuntu 14.10. All presented execution times are CPU times, *i.e.*, only time periods spent in allocated CPUs, which were measured with the `times` system call (POSIX system).

6.1. Experimental Objectives

The experiments aim to answer two research questions:

RQ1 **(sanity check)** what results do the proposed CEGIO algorithms obtain when searching for the functions optimal solution?

RQ2 **(performance)** what is the proposed CEGIO algorithms performance if compared to genetic algorithm, particle swarm, pattern search, simulated annealing, and non-linear programming?

6.2. Description of Benchmarks

In order to answer these research questions, we consider 30 reference functions of global optimization problems extracted from the literature [68]; all reference functions are multivariable with two decision variables. Those functions present different formats, *e.g.*, polynomials, sine, cosine, floor, sum, square root; and can be continuous, differentiable, separable, non-separable, scalable, non-scalable, uni-modal, and multi-modal. Note that the boundaries of these global constrained optimization problems are literally extracted from the literature [68]. Naturally, a large domain will penalize the proposed method performance, but the global optimization accuracy still holds. In real-world applications, the boundaries are very particular and depend on each problem and application. In future, we intend to automatically extract those boundaries using abstract interpretation based on interval, octagons, and polyhedral constrains [69].

The employed benchmark suite is described in Table 1 as follows: benchmark name, domain, and global minimum, respectively. In order to perform the experiments with three different CEGIO algorithms, generalized (Alg. 1),

simplified (Alg. 2), and fast (Alg. 3), a set of programs were developed for each function, taking into account each algorithm and varying the solver and the data-type accordingly. For the experiment with the generalized algorithm, all benchmarks are employed; for the simplified algorithm, 15 functions are selected from the benchmark suite. By previous observation, we can affirm that those 15 functions are semi-definite positive; lastly, we selected 10 convex functions from the benchmark suite to evaluate the fast algorithm.

Table 1: *Benchmark Suite for Global Optimization Problems. For all* $i = 1, 2$.

#	Benchmark	Domain	Global Minima
01	Engvall	$-10 \leq x_i \leq 10$	$f(1,0) = 0$
02	Tsoulos	$-1 \leq x_i \leq 1$	$f(0,0) = -2$
03	Zirilli	$-10 \leq x_i \leq 10$	$f(1.046,0) = -0.3523$
04	Step 2	$-100 \leq x_i \leq 100$	$f(0,0) = 0$
05	Scahffer 4	$-10 \leq x_i \leq 10$	$f(0,1.253) = 0.292$
06	Adjiman	$-1 \leq x_i \leq 2$	$f(2,0.10578) = -2.02181$
07	Cosine	$-1 \leq x_i \leq 1$	$f(0,0) = -0.2$
08	S2	$-5 \leq x_i \leq 5$	$f(x_1,0.7) = 2$
09	Styblinski Tang	$-5 \leq x_i \leq 5$	$f(2.903,2.903) = -78.332$
10	Trecanni	$-5 \leq x_i \leq 5$	$f(\{0,0\},\{2,0\}) = 0$
11	Ursem 1	$-3 \leq x_i \leq 3$	$f(1.697136,0) = -4.8168$
12	Branin RCOS	$-5 \leq x_i \leq 15$	$f(\{-\pi,12.275\},\{\pi,2.275\},$ $\{3\pi,2.425\}) = 0.3978873$
13	Wayburn Seader 2	$-500 \leq x_i \leq 500$	$f(\{0.2,1\},\{0.425,1\}) = 0$
14	Alpine 1	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
15	Egg Crate	$-5 \leq x_i \leq 5$	$f(0,0) = 0$
16	Himmeblau	$-5 \leq x_i \leq 5$	$f(3,2) = 0$
17	Leon	$-2 \leq x_i \leq 2$	$f(1,1) = 0$
18	Price 4	$-10 \leq x_i \leq 10$	$f\{(0,0),(2,4),$ $(1.464,-2.506)\} = 0$
19	Schuwefel 2.25	$-10 \leq x_i \leq 10$	$f(1,1) = 0$
20	Sphere	$0 \leq x_i \leq 10$	$f(0,0) = 0$
21	Booth	$-10 \leq x_i \leq 10$	$f(1,3) = 0$
22	Chung	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
23	Cube	$-10 \leq x_i \leq 10$	$f(1,1) = 0$
24	Dixon & Price	$-10 \leq x_i \leq 10$	$f(x_i) = 0, x_i = 2^{-((2i-2)/2i)}$
25	Power Sum	$-1 \leq x_i \leq 1$	$f(0,0) = 0$
26	Schumer	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
27	Sum Square	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
28	Matyas	$-10 \leq x_i \leq 10$	$f(0,0) = 0$
29	Rotated Ellipse	$-500 \leq x_i \leq 500$	$f(0,0) = 0$
30	Zettl	$-5 \leq x_i \leq 10$	$f(0.029,0) = -0.0037$

For the experiments execution with the proposed algorithms, random values are generated, belonging to the solutions space of each function, and they are used as initialization of the proposed algorithms, as described in Section 4.

Table 2: *Experimental Times Results with the Generic Algorithm (in seconds).*

#	Boolector	Z3	MathSAT	#	Boolector	Z3	MathSAT
1	1020	3653	**662**	16	4495	11320	**10**
2	**305**	9023	2865	17	269	1254	**4**
3	**383**	720	662	18	16049*	110591	**6**
4	3	**1**	11	19	2972	5489	**7**
5	**6785**	14738	33897	20	**1**	**1**	2
6	**665**	2969	19313	21	2660	972	**5**
7	**393**	2358	3678	22	839*	5812*	**2**
8	32	13	**10**	23	170779*	77684*	**5**
9	1330	19620	**438**	24	36337*	22626*	**8**
10	**76**	269	2876	25	**3**	40	4
11	808	**645**	11737	26	445	20	**4**
12	17458	25941	**3245**	27	41	**1**	3
13	33794*	36324	**37**	28	5945	5267	**23**
14	**537**	6788	590	29	1210	2741	**16**
15	5770	3565	**500**	30	271	611	**11**

The other optimization techniques used for comparison, had all benchmarks performed by means of the Optimization Toolbox in MATLAB 2016b [70] with the entire benchmark suite. The time presented in the following tables are related to the average of 20 executions for each benchmark; the measuring unit is always in seconds based on the CPU time.

6.3. Experimental Results

In the next subsections, we evaluate the proposed CEGIO algorithms performance; we also compare them to other traditional techniques. All functions in Table 1 were used for Alg. 1 experiments, functions (#13 $-$ #27) were used for Alg. 2 experiments and functions (#21 $-$ #30) for Alg 3. Furthermore, for all experiments, the minimum-improvement cost function value (δ) and the precision of the decision variables are set to 10^{-4} and 10^{-3}, respectively.

6.3.1. Generalized Algorithm (CEGIO-G) Evaluation

The experimental results presented in Table 2 are related to the performance evaluation of the Generalized Algorithm (CEGIO-G) (cf. Alg. 1). Here, the CPU time is measured in seconds to find the global minimum using the ESBMC tool with a particular SMT solver. Each column of Table 2 is described as follows: columns 1 and 5 are related to functions of the benchmark suite; columns 2 and 6 are related to the configuration of ESBMC with Boolector; columns 3 and 7 are related to ESBMC with Z3; and columns 4 and 8 are related to ESBMC with MathSAT.

All benchmarks are employed for evaluating the generalized algorithm performance. The correct global minima is found in all benchmarks using different SMT solvers: MathSAT, Z3, and Boolector. For all evaluated benchmarks, MathSAT is 4.6 and 3.7 times faster than Z3 and Boolector, respectively,

although there are benchmarks in which MathSAT took longer, *e.g.*, in *Scahffer* (#5) and *Adjiman* (#6) functions. If we compare the performance of MathSAT with other SMT solvers, we can also check that it is routinely faster than Boolector and Z3, which performed better in 60% of the benchmarks, while for Boolector in 30% and Z3 in 13% of all considered benchmarks. In particular, for the*Sphere* (#20) function, Boolector and Z3 performed similarly. Furthermore, the MathSAT solver achieved the best overall performance, considering the overall optimization time.

Initially, all experiments were performed using `float`-type variables, but we noticed that there was either overflow in some particular benchmarks, *e.g.*, the *Cube* (#23) functions. It occurs due to truncation in some arithmetic operations and series, *e.g.*, sines and cosines, once the verification engine employs fixed-point for computations. This might lead to a serious problem if there are several operations being performed with very large inputs, in a way that causes errors that can be propagated; those errors thus lead to incorrect results. For this specific reason, we decided to use `double`-type variables for these particular benchmarks to increase precision. We observed that the global minimum value is always found using double precision, but it takes longer than using `float`-type variables. The cells with asterisks in Table 2 identify the benchmarks that we use `double`- instead of `float`-type.

Additionally, we observed that when the function has more than one global minimum, *e.g.*, *Wayburn Seader* 2 (#13), which has two for the decision variables $f\{(0.2.1), (0.425, 1)\}$, the algorithm first finds the global minimum with the decision variables of less precision, then in this case $f(0.2, 1)$. Analyzing Alg. 1, when an overall minimum value is found, the condition in line 10 is not satisfied, since there is no candidate value less than the current one found; thus, the precision is updated and the outer loop starts again. Even if there is another overall minimum in this new precision, it will not be considered by the ASSUME directive in line 8, since the decision variables define a δ-minor candidate value due to the condition in line 13 of Alg. 1, which corresponds to Eq. 5. In order to find the other global minimum, it would be required to limit it with the ASSUME directive, thus disregarding the previous minimum.

6.3.2. Simplified Algorithm (CEGIO-S) Evaluation

The simplified algorithm (CEGIO-S) is applied to functions that contain invariants about the global minimum, *e.g.*, semi-definite positive functions, where it is not needed to search for their minimum in the f negative values. For instance, the *Leon* (#17) function presented in Eq. (12) has the global minimum at $f(1, 1) = 0$ as follows

$$f(x_1, x_2) = 100(x_2 - {x_1}^2)^2 + (1 - x_1)^2. \tag{12}$$

By inspection it is possible to claim that there are no negative values for $f(x)$. Therefore, in order to effectively evaluate Alg 2, 15 benchmarks are selected, functions #13 − #27 in Table 1, which have modules or exponential pair, *i.e.*, the lowest possible value to global minimum is a non-negative value. The experiments are performed using the `float` data-type, and `double` as needed to avoid overflow, using the same solvers as described in Subsection 6.3.1.

In addition, after exhaustive evaluation with our benchmarks, the learning rate, α, was fixed at 5 empirically for all functions evaluated by this algorithm. According to the experimental results shown in Table 3, we confirmed that all obtained results match those described in the literature [68].

Table 3: *Experimental Results with the Simplified Algorithm (in seconds).*

#	CEGIO-S			CEGIO-G		
	Boolector	Z3	MathSAT	Boolector	Z3	MathSAT
13	215	2446	**30**	33794*	36324	37
14	74	**2**	413	537	6788	590
15	34	**2**	240	5770	3565	500
16	**1**	**1**	6	4495	11320	10
17	1	**<1**	2	269	1254	4
18	**<1**	**<1**	5	16049*	110591	6
19	**1**	2	5	2972	5489	7
20	**<1**	**<1**	**<1**	1	1	2
21	**<1**	**<1**	1	2660	972	5
22	**<1**	**<1**	**<1**	839*	5812*	2
23	**<1**	**<1**	2	170779*	77684*	5
24	14	**2**	6	36337*	22626*	8
25	**<1**	**<1**	2	3	40	4
26	**<1**	**<1**	1	445	20	4
27	**<1**	**<1**	**<1**	41	1	3

Additionally, we can see that the simplified algorithm reduces the optimization time considerably, with particular benchmarks reaching less than 1 second. However, the reduction with the MathSAT solver is less expressive since it models `float`-type variables using floating-point arithmetic in both CEGIO-S and CEGIO-G algorithms, while Boolector and Z3 uses fixed-point arithmetic. We conclude that either our fast algorithm is suitable for fixed-point architectures or MathSAT implements more aggressive simplifications than Boolector and Z3 .

The purpose of this algorithm is to find the global minimum to reduce the verification time, for functions that have invariants about the global minimum. However, the simplified algorithm run-time might be longer than the generalized one since it requires parameter settings according to the function. As described in Subsection 5.1, in line 7 of Alg 2, we have the variable γ that defines the state-space search segmentation; γ is obtained by the difference of the current $f(x)$ and the boundary that we know, divided by the variable α (previously established). If we have a very large absolute value for α, then we would have additional checks, thus creating many more properties to be checked by the verifier (and thus leading it to longer verification times).

If we analyze *S2* (#08) function in Eq. (13), then we can easily inspect that there is no $f(x)$ less than 2; in this case, therefore, in line 1 of Alg 2, one can establish f_m with the value 2. This slightly change in the initialization of f_m in

Alg 2 prunes the state-space search and the verification time accordingly.

$$f(x_1, x_2) = 2 + (x_2 - 0.7)^2 \tag{13}$$

6.3.3. Fast Algorithm (CEGIO-F) Evaluation

The experimental results for the fast algorithm (CEGIO-F) are presented in Table 4. This algorithm is applied to convex functions, where there is only a global minimum; in particular, the state-space is reduced on each iteration of the outer (`for`) loop in Alg 3, ensuring that the global minimum is in the new (delimited) space, and then it performs a new search in that space to reduce the overall optimization time.

In order to evaluate the effectiveness of Alg 3, we selected approximately 10 convex functions of the benchmark suite, functions #21 − #30 in Table 1; we also compare the fast algorithm (CEGIO-F) results with the generalized one (CEGIO-G). We observed that there are significant performance improvements if we compare CEGIO-F to CEGIO-G for convex function benchmarks, *i.e.*, CEGIO-F algorithm is 1000 times faster using the SMT solver Boolector and 750 times faster using the SMT solver Z3 than the (original) CEGIO-G algorithm, as shown in Table 4.

Table 4: *Experimental Results with the Fast Algorithm (in seconds).*

#	CEGIO-F		CEGIO-G	
	Boolector	Z3	Boolector	Z3
21	**<1**	**<1**	2660	972
22	33*	**26***	839*	5812*
23	43*	**25**	170779*	77684*
24	59*	**10***	36337*	22626*
25	**1**	10	3	40
26	**1**	2	445	20
27	1	**<1**	41	1
28	7	**2**	5945	5267
29	2	**1***	1210	2741
30	**63***	76	271	611

6.3.4. Comparison to Other Traditional Techniques

In this section, our CEGIO algorithms are compared to other traditional optimization techniques: genetic algorithm (GA), particle swarm (ParSwarm), pattern search (PatSearch), simulated annealing (SA), and nonlinear programming (NLP).

Table 5 describes the hit rates and the mean time for each function w.r.t. our proposal (ESBMC) and other existing techniques (GA, ParSwarm, PatSearch, SA, and NLP). An identification for each algorithm is defined: (1) Generalized, (2) Simplified, and (3) Fast. All traditional optimization techniques are executed 20 times using MATLAB, for obtaining the correctness rate and the mean time for each function. These techniques are also configured through the default MATLAB settings.

Our hit rate is omitted for the sake of space, but our algorithms have found the correct global minima in 100% of the experiments, considering the precision of 10^{-3} for decision variables, the same criterion was used for analysis Of the other algorithms. The experiments show that our hit rate is superior than any other optimization technique, although the optimization time is usually longer.

The other optimization techniques are very sensitive to non-convexity; for this reason, they are usually trapped by local minima. The other optimization techniques presented better performance in convex functions. Specifically, they converge faster to the response and there are no local minimums that could lead to incorrect results, whereas with the non-convex functions, their hit rate is lower, precisely because there are local minimums.

Table 5: *Experimental Results with the Traditional Techniques and our Best CEGIO Algorithm (in seconds).*

#	ESBMC	GA		ParSwarm		PatSearch		SA		NLP	
	T	R%	T	R%	T	R%	T	R%	T	R%	T
1	$661^{(1)}$	90	1	100	2	90	3	95	1	100	7
2	$305^{(1)}$	100	9	95	1	100	3	75	9	0	6
3	$383^{(1)}$	100	9	100	1	100	3	60	1	75	2
4	$1^{(1)}$	0	9	0	1	0	2	0	8	0	1
5	$6785^{(1)}$	30	1	15	<1	0	<1	0	2	0	<1
6	$665^{(1)}$	0	10	100	1	0	4	80	2	95	2
7	$393^{(1)}$	100	9	100	1	95	3	95	2	15	2
8	$10^{(1)}$	65	<1	100	<1	100	<1	85	1	100	<1
9	$438^{(1)}$	100	9	100	1	50	3	100	1	35	2
10	$76^{(1)}$	0	9	0	1	0	3	0	1	0	2
11	$645^{(1)}$	100	9	100	1	100	3	80	1	65	2
12	$3245^{(1)}$	100	8	100	9	100	4	75	8	0	5
13	$30^{(2)}$	100	1	95	1	100	3	100	1	100	2
14	$2^{(2)}$	25	1	45	3	10	4	50	1	0	9
15	$2^{(2)}$	100	9	100	1	70	3	100	1	25	2
16	$1^{(2)}$	60	9	50	1	25	3	15	1	35	2
17	$< 1^{(2)}$	90	1	75	2	0	7	10	1	100	4
18	$< 1^{(2)}$	0	9	10	2	0	7	0	4	50	2
19	$1^{(2)}$	100	1	95	1	100	3	100	1	100	2
20	$< 1^{(2)}$	100	10	100	7	100	4	100	1	100	2
21	$< 1^{(3)}$	100	10	100	2	100	6	95	1	100	2
22	$26^{(3)}$	100	9	100	1	100	4	90	1	100	5
23	$25^{(3)}$	20	1	30	3	0	8	10	2	100	7
24	$10^{(3)}$	0	9	0	2	0	3	0	1	0	2
25	$1^{(3)}$	100	9	100	1	100	3	50	1	100	2
26	$1^{(3)}$	100	9	100	1	100	4	75	1	100	4
27	$< 1^{(3)}$	100	9	100	1	100	4	100	1	100	2
28	$2^{(3)}$	100	9	100	1	100	8	10	1	100	2
29	$1^{(3)}$	100	9	100	2	100	7	100	1	100	2
30	$63^{(3)}$	100	9	100	1	100	3	60	1	75	2

7. Additional Performance Issues

In the previous Section, the proposed algorithms were evaluated using a large set of benchmarks, and presented competitive performance if compared to traditional non-convex optimization techniques. However, some issues related to the efficiency of the proposed algorithm must be discussed:

- the effect of a large amount of decision variables;
- the effect of the variation of minimum variation step;

- the use of special trigonometric functions, *e.g.*, `sine` and `cosine`.

7.1. Effects of varying the number of decision variables and the minimum variation step

In Section 6, all the evaluated benchmarks were defined by two-dimensional functions and by using minimum variation step (δ) of 10^{-4}. In this subsection, we will briefly discuss the effect of the variation of these characteristics in the optimization time.

In order to exploit the effect of the variables number in the proposed algorithms, the Styblinski-Tang (#09) function was chosen, which is given by Eq. (14)

$$f(x_1, x_2, ..., x_n) = \frac{1}{2}\sum_{i=2}^{n}(x_i^4 - 16x_i^2 + 5x_i), \tag{14}$$

where n is the number of decision variables.

The generalized algorithm (Alg. 1) and the MathSAT solver were used to evaluate this particular non-convex function. We use the MathSAT solver since it showed a better performance in previous experiments if compared to Z3 and Boolector. In addition, the minimum improvement at each iteration in the cost function, δ, varies from 10^{-1} to 10^{-4} and the number of variables from 2 to 10. Figure 6 shows the optimization times for the described configurations.

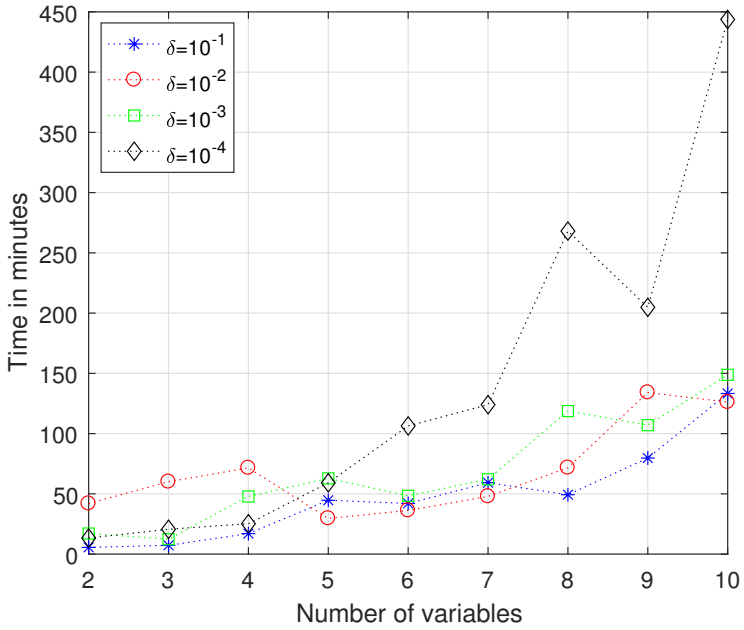

Figure 6: *Effect on the number of decision variables to the optimization time, for different values of δ.*

Note that, the optimization time tends to increase if the number of variables increases. Furthermore, the optimization time also tends to be longer if δ gets

smaller. These results were expected once that the reduction of δ tends to increase the number of algorithms iterations. Note that the worst case size of the search space can be estimated (by analysis of the CEGIO-G, Algorithm 1) at $\prod_{i=1}^{k} ||Domain(x_i)||$.

The worst case of CEGIO-G can be evaluated through the previous estimative, however, each algorithm step generally finishes before the exploration of all the search domain once it stops when the $\neg l_{suboptimal}$ (defined in Eq. (5)) is `SAT`. For this reason, even when the number of variables is increased and the δ is decreased, the optimization time might reduce. Figure 6 shows that there are cases where the optimization time decreases significantly with the increasing of the number of variables.

7.2. Special Operations

Another important issue related to the efficiency of the proposed algorithms is related to the implementation of mathematical functions and its relation with optimization accuracy and time. In particular, the trigonometric functions, *e.g.*, `sin()` and `cos()`, in operations related to the cost function evaluation of the optimization problem, require special treatment. SMT solvers do not support this type of operations, however, we can use the `math.h` library of the C programming language for this purpose, and the results of these operations (performed via `math.h` implementation) are used to compute the cost function. However, the use of the `math.h` library makes the SMT solver to solve a harder verification condition, which usually results in a long optimization time and efficiency loss. To avoid this problem, `sin()` and `cos()` functions have been reimplemented in a library called `math2.h`, using a set of assumptions and based on Taylor series [71]. Our implementation presents better performance than the original functions of the `math.h` library if they performed in verification, while maintaining the required accuracy. All those reimplemented functions have the precision of 10^{-6}.

Note also that in all algorithms, the cost function is computed via the `float` variable, x, which is defined as, $x = X/10^{\epsilon}$, then the manipulation of the `integer` variable, X, to achieve the desired precision, does not influence in the cost function calculation.

8. Conclusions

This paper presented three variants of a counterexample guided inductive optimization approach for optimizing a wide range of functions based on counterexamples extracted from SMT solvers. In particular, this work proposed algorithms to perform inductive generalization based on counterexamples provided by a verification oracle for optimizing convex and non-convex functions and also presented respective proofs for global convergence. Furthermore, the present study provided an analysis about the influence of the solver and data-types in the performance of the proposed algorithms.

All proposed algorithms were exhaustively evaluated using a large set of public available benchmarks. We also evaluated the present algorithms

performance using different SMT solvers and compared them to other state-of-art optimization techniques (genetic algorithm, particle swarm, pattern search, nonlinear programming, and simulated annealing). The counterexample guided inductive optimization algorithms are able to find the global optima in 100% of the benchmarks, and the optimization time is significantly reduced if compared to Araújo *et al.* [13]. Traditional optimization techniques are typically trapped by local minima and are unable to ensure the global optimization, although they still present lower optimization times than the proposed algorithms.

In contrast to previous optimization techniques, the present approaches are suitable for every class of functions; they are also complete, providing an improved accuracy compared to other existing traditional techniques. Future studies include the application of the present approach to autonomous vehicles navigation systems, enhancements in the model-checking procedure for reducing the verification time by means of multi-core verification [26] and invariant generation [64, 72]. We also intend to improve the proposed algorithms as follows: i) implement δ to be computed dynamically, since it is currently fixed in the proposed algorithm, especially for Alg. 2, which depends on a dynamic learning rate (α); ii) implementing heuristic to reduce the optimization time; iii) use powers of 2 instead of powers of 10 in divisions for space discretization, thus improving the numerical performance. Finally, we intend to extend all present approaches for multi-objective optimization problems.

9. Acknowledgments

The authors thank the financial support of FundaÃğÃčo de Amparo Ãă Pesquisa do Estado do Amazonas (FAPEAM), Brazil, the Brazilian National Research Council (CNPq), and the Coordination for the Improvement of Higher Education Personnel (CAPES).

References

[1] K. Deb, Optimization for Engineering Design: Algorithms and Examples, Prentice-Hall of India, 2004.

[2] D. K. Gattie, R. C. Wicklein, Curricular value and instructional needs for infusing engineering design into k-12 technology education, Journal of Technology Education 19 (1) (2007) 13. doi:10.21061/jte.v19i1.a.1.

[3] Y. Shoham, Computer science and game theory, Commun. ACM 51 (8) (2008) 74–79. doi:10.1145/1378704.1378721.

[4] J. Teich, Hardware/software codesign: The past, the present, and predicting the future, Proceedings of the IEEE 100 (Special Centennial Issue) (2012) 1411–1430. doi:10.1109/JPROC.2011.2182009.

[5] M. Kowalski, C. Sikorski, F. Stenger, Selected Topics in Approximation and Computation, Oxford University Press, 1995.

[6] U. Derigs, Optimization and Operations Research – Volume I:, EOLSS Publications, 2009.

[7] R. Garfinkel, G. Nemhauser, Integer programming, Series in decision and control, Wiley, 1972.

[8] M. Bartholomew-Biggs, The Steepest Descent Method, Springer US, Boston, MA, 2008, Ch. 7, pp. 1–8. doi:10.1007/978-0-387-78723-7_7.

[9] D. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Artificial Intelligence, Addison-Wesley Publishing Company, 1989.

[10] M. Cavazzuti, Optimization Methods: From Theory to Design Scientific and Technological Aspects in Mechanics, Springer Berlin Heidelberg, 2012.

[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: 2013 Formal Methods in Computer-Aided Design, 2013, pp. 1–8. doi:10.1109/FMCAD.2013.6679385.

[12] M. Dorigo, M. Birattari, T. Stutzle, Ant colony optimization, IEEE Comput. Intell. Mag. 1 (4) (2006) 28–39. doi:10.1109/MCI.2006.329691.

[13] R. Araújo, I. Bessa, L. Cordeiro, J. E. C. Filho, SMt-based verification applied to non-convex optimization problems, in: Proceedings of VI Brazilian Symposium on Computing Systems Engineering, 2016, pp. 1–8. doi:10.1109/SBESC.2016.010.

[14] L. De Moura, N. Bjørner, Z3: An efficient SMT solver, in: TACAS, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340.

[15] R. Brummayer, A. Biere, Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2009, pp. 174–177.

[16] A. Cimatti, A. Griggio, B. Schaafsma, R. Sebastiani, The mathSAT5 SMT solver, in: Tools and Algorithms for the Construction and Analysis of Systems, 2013, pp. 93–107. doi:10.1007/978-3-642-36742-7.

[17] M. Jamil, X.-S. Yang, A literature survey of benchmark functions for global optimisation problems, IJMMNO 4 (2) (2013) 150–194. doi:10.1504/IJMMNO.2013.055204.

[18] A. Olsson, Particle Swarm Optimization: Theory, Techniques and Applications, Engineering tools, techniques and tables, Nova Science Publishers, 2011.

[19] P. Alberto, F. Nogueira, H. Rocha, L. N. Vicente, Pattern search methods for user-provided points: Application to molecular geometry problems, SIAM Journal on Optimization 14 (4) (2004) 1216–1236.

[20] P. J. M. Laarhoven, E. H. L. Aarts (Eds.), Simulated Annealing: Theory and Applications, Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[21] R. H. Byrd, J. C. Gilbert, J. Nocedal, A trust region method based on interior point techniques for nonlinear programming, Mathematical Programming 89 (1) (2000) 149–185.

[22] R. Nieuwenhuis, A. Oliveras, On SAT modulo theories and optimization problems, in: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing, SAT'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 156–169. doi:10.1007/11814948_18.

[23] A. Nadel, V. Ryvchin, Bit-Vector Optimization, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 851–867. doi:10.1007/978-3-662-49674-9_53.

[24] H. Eldib, C. Wang, An SMT based method for optimizing arithmetic computations in embedded software code, IEEE CAD 33 (11) (2014) 1611–1622. doi:10.1109/TCAD.2014.2341931.

[25] G. G. Estrada, A note on designing logical circuits using SAT, in: ICES, Springer Berlin Heidelberg, 2003, pp. 410–421. doi:10.1007/3-540-36553-2_37.

[26] A. Trindade, H. Ismail, L. Cordeiro, Applying multi-core model checking to hardware-software partitioning in embedded systems, in: SBESC, 2015, pp. 102–105. doi:10.1109/SBESC.2015.26.

[27] A. Trindade, L. Cordeiro, Aplicando verificação de modelos para o particionamento de hardware/software, in: SBESC, 2014, p. 6.
URL http://sbesc.lisha.ufsc.br/sbesc2014/dl185

[28] A. Trindade, L. Cordeiro, Applying SMT-based verification to hardware/-software partitioning in embedded systems, DES AUTOM EMBED SYST 20 (1) (2016) 1–19. doi:10.1007/s10617-015-9163-z.

[29] S. Cotton, O. Maler, J. Legriel, S. Saidi, Multi-criteria optimization for mapping programs to multi-processors, in: SIES, 2011, pp. 9–17.

[30] 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: 55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016, 2016, pp. 6683–6688. doi:10.1109/CDC.2016.7799298.

[31] M. T. M. Pister, A. Bauer, Tool-support for the analysis of hybrid systems and models, 2007 10th Design, Automation and Test in Europe Conference and Exhibition 00 (2007) 172. doi:10.1109/DATE.2007.364411.

[32] 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).

[33] Y. Shoukry, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, P. Tabuada, SMC: Satisfiability modulo convex optimization, in: Proceedings of the 20th ACM International Conference on Hybrid Systems: Computation and Control, HSCC '17, ACM, New York, NY, USA, 2017, p. 10. doi:10.1145/3049797.3049819.

[34] N. Bjørner, A.-D. Phan, L. Fleckenstein, νZ - an optimizing SMT solver, in: TACAS, Springer Berlin Heidelberg, 2015, pp. 194–199. doi:10.1007/978-3-662-46681-0_14.

[35] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, M. Chechik, Symbolic optimization with SMT solvers, in: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, ACM, New York, NY, USA, 2014, pp. 607–618. doi:10.1145/2535838.2535857.

[36] R. Sebastiani, P. Trentin, OptiMathSAT: A tool for optimization modulo theories, in: CAV, Springer International Publishing, 2015, pp. 447–454. doi:10.1007/978-3-319-21690-4_27.

[37] R. Sebastiani, S. Tomasi, Optimization modulo theories with linear rational costs, ACM TOCL 16 (2) (2015) 12:1–12:43. doi:10.1145/2699915.

[38] R. Sebastiani, P. Trentin, Pushing the envelope of optimization modulo theories with linear-arithmetic cost functions, in: TACAS, Springer Berlin Heidelberg, 2015, pp. 335–349. doi:10.1007/978-3-662-46681-0_27.

[39] Z. Pavlinovic, T. King, T. Wies, Practical SMT-based type error localization, in: ICFP, 2015, pp. 412–423.

[40] E. A. Galperin, Problem-method classification in optimization and control, Computers & Mathematics with Applications 21 (6âĂŞ7) (1991) 1 – 6. doi:10.1016/0898-1221(91)90155-W.

[41] J.-B. Hiriart-Urruty, Conditions for Global Optimality, Springer US, Boston, MA, 1995, pp. 1–26. doi:10.1007/978-1-4615-2025-2_1.

[42] A. Torn, A. Zilinskas, Global Optimization, Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[43] J. Hu, Y. Wang, E. Zhou, M. C. Fu, S. I. Marcus, A Survey of Some Model-Based Methods for Global Optimization, Birkhäuser Boston, Boston, 2012, pp. 157–179. doi:10.1007/978-0-8176-8337-5_10.

[44] D. R. Jones, A taxonomy of global optimization methods based on response surfaces, Journal of Global Optimization 21 (4) (2001) 345–383. doi:10.1023/A:1012771025575.

[45] C. Floudas, Deterministic Global Optimization, Nonconvex Optimization and Its Applications, Springer, 2000.

[46] J. Snyman, Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms, Applied Optimization, Springer, 2005.

[47] D. Scholz, Deterministic Global Optimization: Geometric Branch-and-bound Methods and their Applications, Springer Optimization and Its Applications, Springer New York, 2011.

[48] N. V. Findler, C. Lo, R. Lo, Pattern search for optimization, Mathematics and Computers in Simulation 29 (1) (1987) 41 – 50. doi:10.1016/0378-4754(87)90065-6.

[49] K. Marti, Stochastic Optimization Methods, Springer, 2005.

[50] S. Gao, S. Kong, E. M. Clarke, dReal: An SMT Solver for Nonlinear Theories over the Reals, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, Ch. 4, pp. 208–214. doi:10.1007/978-3-642-38574-2_14.

[51] E. M. Clarke, Jr., O. Grumberg, D. A. Peled, Model Checking, MIT Press, Cambridge, MA, USA, 1999.

[52] C. Baier, J.-P. Katoen, Principles of Model Checking (Representation and Mind Series), The MIT Press, 2008.

[53] D. Beyer, Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016), Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, Ch. 7, pp. 887–904. doi:10.1007/978-3-662-49674-9_55.

[54] A. Biere, Bounded model checking, in: Handbook of Satisfiability, IOS Press, 2009, pp. 457–481.

[55] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, Satisfiability modulo theories, in: Handbook of Satisfiability, IOS Press, 2009, pp. 825–885.

[56] L. Cordeiro, B. Fischer, J. Marques-Silva, SMT-based bounded model checking for embedded ANSI-C software, IEEE TSE 38 (4) (2012) 957–974. doi:10.1109/TSE.2011.59.

[57] D. Kroening, M. Tautschnig, CBMC – C Bounded Model Checker, Springer Berlin Heidelberg, 2014, pp. 389–391. doi:10.1007/978-3-642-54862-8_26.

[58] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, B. Fischer, ESBMC 1.22 - (competition contribution), in: TACAS, 2014, pp. 405–407.

[59] B. R. Abreu, Y. M. R. Gadelha, C. L. Cordeiro, B. E. de Lima Filho, S. W. da Silva, Bounded model checking for fixed-point digital filters, JBCS 22 (1) (2016) 1–20. doi:10.1186/s13173-016-0041-8.

[60] I. V. Bessa, H. I. Ismail, L. C. Cordeiro, J. E. C. Filho, Verification of fixed-point digital controllers using direct and delta forms realizations, DES AUTOM EMBED SYST 20 (2) (2016) 95–126. doi:10.1007/s10617-016-9173-5.

[61] I. V. d. Bessa, H. I. Ismail, L. C. Cordeiro, J. E. C. Filho, Verification of Delta Form Realization in Fixed-Point Digital Controllers Using Bounded Model Checking, in: SBESC, 2014, pp. 49–54. doi:10.1109/SBESC.2014.14.

[62] IEEE, IEEE standard for binary floating-point numbers.

[63] D. Beyer, M. E. Keremoglu, CPAchecker: A tool for configurable software verification, in: CAV, Springer Berlin Heidelberg, 2011, pp. 184–190. doi:10.1007/978-3-642-22110-1_16.

[64] 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. doi:10.1007/s10009-015-0407-9.
URL http://dx.doi.org/10.1007/s10009-015-0407-9

[65] L. Li, Selected Applications of Convex Optimization, Springer Optimization and Its Applications, Springer Berlin Heidelberg, 2015.

[66] S. Boyd, L. Vandenberghe, Convex Optimization, Cambridge University Press, New York, NY, USA, 2004.

[67] P. Pereira, H. Albuquerque, H. Marques, I. Silva, C. Carvalho, L. Cordeiro, V. Santos, R. Ferreira, Verifying CUDA programs using SMT-based context-bounded model checking, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, ACM, New York, NY, USA, 2016, pp. 1648–1653. doi:10.1145/2851613.2851830.
URL http://doi.acm.org/10.1145/2851613.2851830

[68] 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

[69] W. Rocha, H. Rocha, H. Ismail, L. C. Cordeiro, B. Fischer, Depthk: A k-induction verifier based on invariant inference for C programs - (competition contribution), in: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II, Vol. 10206 of LNCS, 2017, pp. 360–364. doi:10.1007/978-3-662-54580-5_23.

[70] The Mathworks, Inc., Matlab Optimization Toolbox User's Guide (2016).

[71] W. Press, Numerical Recipes 3rd Edition: The Art of Scientific Computing, Cambridge University Press, 2007.

[72] H. Rocha, H. Ismail, L. Cordeiro, R. Barreto, Model checking embedded c software using k-induction and invariants, in: Proceedings of VI Brazilian Symposium on Computing Systems Engineering, 2015, pp. 90–95. doi:10.1109/SBESC.2015.24.