Lattice-based SMT for Program Verification

Karine Even-Mendoza
King's College London, UK
karine.even_mendoza@kcl.ac.uk

Antti E. J. Hyvärinen
Università della Svizzera italiana, Switzerland
antti.hyvaerinen@usi.ch

Hana Chockler
King's College London, UK
hana.chockler@kcl.ac.uk

Natasha Sharygina
Università della Svizzera italiana, Switzerland
natasha.sharygina@usi.ch

***Abstract*—We present a lattice-based satisfiability modulo theory for verification of programs with library functions, for which the mathematical libraries supporting these functions contain a high number of equations and inequalities. Common strategies for dealing with library functions include treating them as uninterpreted functions or using the theories under which the functions are fully defined. The full definition could in most cases lead to instances that are too large to solve efficiently.**

Our lightweight theory uses lattices for efficient representation of library functions by a subset of guarded literals. These lattices are constructed from equations and inequalities of properties of the library functions. These subsets are found during the lattice traversal. We generalise the method to a number of lattices for functions whose values depend on each other in the program, and we describe a simultaneous traversal algorithm of several lattices, so that a combination of guarded literals from all lattices does not lead to contradictory values of their variables.

We evaluate our approach on benchmarks taken from the robotics community, and our experimental results demonstrate that we are able to solve a number of instances that were previously unsolvable by existing SMT solvers.

I. Introduction

The satisfiability modulo theories (SMT) [1] reasoning framework is currently one of the most successful approaches to verifying software in a scalable way. The approach is based on modelling the software and its specifications in propositional logic, while expressing domain-specific knowledge with first-order theories connected to the logic through equalities. Successful verification of software relies on finding a model that is expressive enough to capture software behaviour relevant to correctness, while being sufficiently high-level to prevent reasoning from becoming prohibitively expensive.

Finding a scalable way for verifying programs or systems which use library functions as a main part of their application (e.g., implementation of robots' movements in the Robot Operating System (ROS) [2]) is a non-trivial task: the code may contain hundreds of interacting expressions of the properties of the library functions, whose truth values depend on each other. A straightforward solution would be to use increasingly precise theories. However, this approach results in prohibitively expensive computations (e.g., by adding details at the bit-level to describe trigonometric functions, which would be very expensive).

Trigonometric functions serve as a good illustration of the problem outlined above, as many application domains, such as robotics, planning [3], and simulations for physics and engineering [4], rely on the computation of trigonometric functions. Verification of software using trigonometric library functions [5]–[10] either requires a large amount of numerical calculations of polynomials along with irrational numbers, or uses large look-up tables for the trigonometric functions which tend to be less precise and consume memory [11]. The former technique usually replaces the irrational expressions with rational expressions with a defined error bound [5], [6], [12]–[14] in order to bound or to evaluate trigonometric expressions to some precision. A more precise approach relies on Taylor series representation of trigonometric functions over reals. However, this leads to complex computations, and the resulting instances are too large to solve efficiently for all but the smallest programs.

Finally, the solver implemented in HiFrog [15] supports the addition of sets of equations and inequalities as *user-defined function summaries*. We can, therefore, extract the known properties of library functions from the external libraries and encode them as user-defined SMT summaries which are passed to HiFrog, and further, to the SMT solver. However, this approach is not scalable either, as we do not know beforehand which properties are going to be relevant for solving a particular instance. Hence, for library functions with a large number of equations (such as trigonometric functions—there are many equations describing properties of these functions on some subdomains), the user-defined summaries will render the instance too large to be solvable efficiently (or at all).

In this paper, we present a novel approach to reasoning about programs whose correctness depends on the values of library functions. Our approach uses the concept of *subset lattices* to construct an efficient representation of known properties of these functions. Essentially, we order the set of subsets of equations describing properties of library functions in a lattice, where each element corresponds to a set of properties that hold for some subdomain of the inputs. At every iteration of the algorithm, we verify the program with only a subset of equations that corresponds to the current element

in the lattice. If this subset is insufficient for the verification (that is, does not provide enough information about the library function), we *refine* it by traversing the lattice to a higher element, containing a superset of the equations.

This lattice-based counter-example-guided abstraction refinement algorithm (LB-CEGAR) is based on the traditional counter-example-guided abstraction refinement (CEGAR) [16], [17], but replaces the refinement of the theory by the refinement of the set of equations for the library function in the program. Our approach is similar to the traditional CEGAR approach in the sense that a SAT result may indicate a real counterexample (in which case there are concrete values of symbolic variables that show the existence of this execution), or a *spurious* counterexample, where the satisfying assignment provided by the solver is due to overapproximation in the representation of the program. In contrary to the traditional CEGAR, where an UNSAT result indicated that there are no counterexamples in an abstraction of the program and hence in the concrete program as well, in LB-CEGAR the UNSAT result merely means that there are no counterexamples in the current subdomain of the input to the library function. As we describe in the paper, the lattice is constructed so that every *lattice frontier* covers the whole domain of the input variables. Hence, in case of an UNSAT result, the LB-CEGAR algorithm attempts to construct a *frontier of unsatisfiability*. Such a frontier would indicate that there are no counterexamples in the current abstraction for each subdomain of the input, and hence for the whole domain as well.

In our previous work, we described a simplified LB-CEGAR algorithm for the case of one library function in the program and for small lattices [18]. In this work, we extend LB-CEGAR to the general case, where the program may contain several library functions whose values can be interconnected (for example, $\sin x$ and $\cos x$). Furthermore, each function can appear in the program multiple times, thus inducing several instances of the lattice, which are traversed simultaneously. We describe the generalised LB-CEGAR algorithm and analyse its worst-case complexity and heuristics in Sec. IV.

Our results are based on the trigonometric functions being treated as uninterpreted functions in the encoding of the problem to the SMT solver and the encoding of the mathematical equations as user-defined function summaries in the semantics of reals, an approach commonly followed in in modelling software [15], [19]–[21]. We assume the correctness of these equations (such as $\sin^2 x + \cos^2 x = 1$) over real numbers. An alternative approach of verifying programs based on the IEEE floating-point semantics is challenging due to the difficulties stemming from the implementation of the trigonometric functions in the underlying architecture. There are clear advantages in the approach followed in the current paper of pinpointing the subset of mathematical equations that are instrumental for the correctness of the program under verification to the challenge of verification over floating-point semantics. We leave the exploration of this direction to the future work.

The following example illustrates the motivation for LB-CEGAR on a small program with trigonometric functions.

```
#include <math.h>

double nonlin(double x) {
    double x_sin = sin(x);
    double x_cos = cos(x);
    return x_sin*x_sin + x_cos*x_cos;
}

void main() {
    double y = nondet();
    double z = nonlin(y);
    assert(z == 1);
}
```

Figure 1. A program with two different library functions.

Example 1: The program in Fig. 1 contains two library function calls: $\sin x$ and $\cos x$. The correctness of the program follows immediately from the following trigonometric identity:

$$\forall x \in \mathbb{R}. \sin^2 x + \cos^2 x = 1. \tag{1}$$

Clearly, verifying this program with $\sin x$ and $\cos x$ treated as uninterpreted functions (that is, having nondeterministic values) would result in numerous spurious counterexamples. LB-CEGAR overcomes this problem by representing some salient properties of these functions as lattices of equations, including, in particular, Eq. (1).

In this case, the elements of the lattices for $\sin x$ and for $\cos x$ at each iteration of LB-CEGAR are not independent, as Eq. (1) should hold for each combination of these elements. Moreover, having a lattice only for one function would not suffice for proving the correctness of this program, as then we would have Eq. (1) only for one of these functions, while leaving the other one as a non-deterministic variable. This would lead to spurious counter-examples, stemming from assigning illegal values to the non-deterministic variable. For example, if $\cos x$ is left as a non-deterministic variable, it could be assigned the value 2, hence falsifying the assertion.

We implemented the generalised LB-CEGAR algorithm in the bounded model checker HIFROG [15] supporting a subset of the C language, using the SMT solvers OPENSMT [22] and Z3 [23], and evaluated the implementation on a large set of benchmarks containing programs whose correctness depends on the values of trigonometric functions. The experimental results clearly demonstrate an advantage to LB-CEGAR over other approaches.

The presentation is organized as follows. After preliminaries in Sec. II, we present in Sec. III the key insight of lattices for *guarded literals* that are the basic building blocks of our abstraction. In Sec IV we describe the LB-CEGAR algorithm that operates on the guarded literals. The implementation is outlined in Sec. V and the experimental results in Sec. VI. Due to lack of space, the proofs are omitted from this version but are available in the full version of the paper at [24]. The implementation, the set of benchmarks, and the experimental results are available at [24]–[26]. In [24] we also provide examples of lattice construction with different properties of library functions as well as refinement with such lattices of a small code example in C.

II. Preliminaries

A. SMT-based bounded model checking

Let P be a *loop-free* program represented as a transition system, and t a *safety property*, that is, a formula over the variables of P. The bounded model-checking problem amounts to determining whether all states of P, reachable within a predefined bound, satisfy t. More specifically, the task of a model-checker is to find a counterexample, that is, a bounded execution of P that falsifies t, or to prove the absence of such executions.

In the SMT-based bounded model-checking approach followed in this paper, the model-checker encodes all bounded executions of P as an SMT formula, conjoins it with the negation of t, and invokes an SMT solver to check the satisfiability of the resulting formula. If the formula is deemed unsatisfiable, the program is safe, that is, P satisfies t. Otherwise, a satisfying assignment found by the SMT solver is used to build a concrete counter-example. Depending on the theory used by the SMT solver, an abstract counterexample can also be *spurious*, that is, not corresponding to any concrete execution. This situation arises when the theory is too abstract, and hence the resulting overapproximation of the behaviours of the program is too coarse. In this case, the program is re-verified with a more refined theory.

B. Function summaries

The tool HiFrog allows to incorporate function summaries into the verification process [15]. These summaries can be *interpolants* [27] from one of the previous iterations of model-checking or *user-defined* summaries supplied by the user, based on their external knowledge of the system. Some examples of user-defined summaries are available on HiFrog's webpage [28] and in the full version of this paper. We exploit this functionality by providing HiFrog with the library of *user-defined* summaries derived from external libraries for the functions, whose values are critical for determining correctness of the program, and we organise them in lattices as we explain below. This allows us to verify programs in the most abstract theory of equality logic with uninterpreted functions (EUF).

C. A subset lattice

For a given set X, the family of all subsets of X, partially ordered by the inclusion operator, forms a *subset lattice* $SL(X)$. The $\sqcap$ and $\sqcup$ operators are defined on $SL(X)$ as *intersection* and *union*, respectively. The top element $\top$ is the set X, and the bottom element $\bot$ is the empty set $\emptyset$. We note that $SL(X)$ is a De-Morgan lattice [29], as meet and join distribute over each other.

A *meet-semilattice* $\langle L, \sqcap \rangle$ of a lattice L is a partially ordered set (poset) when the $\sqcap$ operator is defined for any subset of its elements (but not necessarily the $\sqcup$ operator). A *subposet* of a lattice is a subset of elements, which follow the same partial order as in the poset. A *chain* of a lattice is a subposet of a lattice where every two elements are ordered.

In this paper, we consider $SL(X)$ and $\langle SL(X), \sqcap \rangle$, for X being a finite set of guarded expressions, as defined in Sec. III.

III. Lattices of Guarded Literals

In this section we describe the construction of lattices of expressions for external functions.

A *guarded literal* is a Boolean expression describing some property of the function in question, together with the guard that defines a *continuous subdomain* of the inputs for which this property holds. For example, the property expressing the fact that for $0 < x < 2$, the value of $\sin x$ is positive is described by the guarded literal

$$(assume(0 < x < 2)) \wedge (\sin x > 0),$$

where $(0 < x < 2)$ is a *guard* (denoted by G) of the *literal* $(\sin x > 0)$. Literals that hold for all x (such as, for example, $(\sin x \leq 1)$ are guarded with $assume(\mathbf{true})$. A guard cannot refer to a non-continuous domain. For example,

$$(assume((0 < x < 2) \vee (7 < x < 8))) \wedge (\sin x > 0)$$

is not a legal guarded literal in our framework.

Given a set of guarded literals F for a library function f, the subset lattice $SL(F)$ consists of all subsets of these literals. However, it is easy to see that some elements in $SL(F)$ contain literals with contradictory guards. For example, a lattice of all subsets of $\sin x$ could contain the element $(assume(0 < x < 2)) \wedge (\sin x > 0)$ and the element $(assume(x = 0)) \wedge (\sin x = 0)$, which do not intersect on any subdomain of x. To reduce the size of the lattice and to avoid unnecessary calls to the SMT solver, we reduce $SL(F)$ to a *meet semi-lattice* $L = \langle SL(F), \sqcap \rangle$ by removing all elements that have contradictory guards (that is, the conjunction of their guards is false).

Note that after the removal of contradictory elements, the resulting set of subsets is no longer closed under union, but it is still closed under intersection, hence the resulting set is a meet semi-lattice. Note also that the resulting meet semi-lattice can have a set of maximal elements instead of the single maximal element. For brevity, in the rest of the paper we refer to the meet semi-lattice of guarded literals for a function f simply as a lattice.

A *frontier* of a lattice L is a set of elements $X(L)$ such that each chain from $\bot$ to a maximal element in L intersects $X(L)$ in at least one element. The LB-CEGAR algorithm described in the next section relies on the observation that *the union of guards of each frontier of the lattice is the whole domain of the inputs*. If this is not the case, we add elements to the lattice to cover the missing subdomains. For the example of $\sin x$ above, if we have only two guarded literals $(assume(0 < x < 2)) \wedge (\sin x > 0)$ and $(assume(x = 0)) \wedge (\sin x = 0)$ in our set, we add the guarded literals $(assume(x < 0)) \wedge \mathbf{true}$ and $(assume(x \geq 2)) \wedge \mathbf{true}$ to the set to cover the whole domain of x (recall that the guards should refer to continuous subdomains, hence we need to add two guarded literals).

Claim 1: If the union of guards of a given set of guarded literals S covers the whole domain of the input, then for each frontier $X(L_S)$ of the subset lattice L_S of S, the union of guards of $X(L_S)$ also covers the whole domain of the input.

And conversely, if the union of guards of a subset $X(L_S)$ of the elements of L_S covers the whole domain of the inputs, then $X(L_S)$ is a frontier of L_S.
Informally, the claim follows from the structure of the subset lattice and the fact that the bottom element of L_S covers the whole domain (the reader is referred to the full version for the formal proof of this claim).

The procedure described in this section is done at the *preprocessing stage*, once for each library function, and the resulting lattices can be used in verification of multiple programs.

IV. The Lattice-based Counterexample Guided Abstraction Refinement (LB-CEGAR) Algorithm

In this section we present the main contribution of the paper, the LB-CEGAR algorithm. We start with an informal overview and then present the formal description of the algorithm. We proceed by discussing its worst-case complexity and then present several heuristics that reduce the complexity for the majority of the cases.

A. Overview of the LB-CEGAR algorithm

The inputs to the Lattice-based Counterexample Guided Abstraction Refinement (LB-CEGAR) algorithm (Alg. 1) are a bounded loop-free program P that includes a function f, and a safety property t. The algorithm follows the standard procedure of translating P and the negation of t to a first-order formula φ and invoking an SMT solver to determine the satisfiability of φ. In contrast to the standard approach, in LB-CEGAR the SMT solver has access, in addition to φ, to the external lattice L_f of guarded literals for f constructed in Sec. III. At each iteration LB-CEGAR adds the set of guarded literals in the current element E of this lattice to φ before sending φ to the SMT solver.

The refinement loop in LB-CEGAR, invoked when a satisfying assignment does not correspond to a concrete counterexample, amounts to the traversal of L_f as described below in the procedure $traverse_{SAT}$.

The algorithm terminates when it either finds a satisfying assignment that corresponds to a concrete counterexample (and hence a bug in P), reaches all maximal elements of L_f without finding concrete counterexamples for any of the satisfying assignments (that is, the current set of properties of f encoded in L_f is insufficient to verify P), or finds a *frontier* of L_f such that φ is unsatisfiable with each element of the frontier separately. The last case implies that there are no counterexamples in the overapproximation of P for the whole domain of the inputs, and hence P satisfies t.

An iteration of LB-CEGAR with a program P, a safety property t, and a current element e consisting of the set of guarded literals $S(e)$ of the lattice L_f results in one of the following cases (for one library function f and a single occurrence of f in the loop-free program P):

- An SMT solver finds a satisfying assignment for φ with $S(e)$, and there is a concrete counterexample corresponding to this assignment. The algorithm terminates, outputting the counterexample as an evidence of the negative result of correctness of P.
- An SMT solver finds a satisfying assignment for φ with $S(e)$, but there is no concrete counterexample corresponding to this assignment. The algorithm invokes a *refinement step* that amounts to traversing L_f to an element e' that refines e, that is, $S(e) \subset S(e')$. If no such element exists, then e is a maximal element of L_f, and the algorithm terminates with inconclusive results.
- An SMT solver returns the UNSAT result for φ with $S(e)$. In other words, there is no satisfying assignment to φ in the subdomain of inputs induced by e. The refinement step of LB-CEGAR is, then, to check satisfiability of φ with elements of L_f that complement the subdomain of e to the whole domain of the input (that is, with elements of L_f that together with e form a *frontier* of L_f).
- An SMT solver returns the UNSAT result for φ with $S(e)$, and e is a part of a frontier of L_f for which φ is unsatisfiable. This result implies that there is no satisfying assignment to φ over the whole domain of the inputs, and therefore P is safe with respect to t.

If the function f appears in P several times, an instance of L_f is created for each occurrence. Furthermore, if P contains more than one library function for which we have a lattice of guarded literals, all these lattices are incorporated in LB-CEGAR. For programs with trigonometric functions, which are the primary domain of application in this paper, it is often the case that an equation includes several functions — see, for example, the program in Ex. 1.

B. The main LB-CEGAR algorithm

We present here the pseudo-code for LB-CEGAR and discuss the general case of several functions and several occurrences of each function in the program. The input to the algorithm is a loop-free program P, a safety property t, and a set of lattices $Lattices$.

The sub-procedures and notations in Alg. 1 are defined as follows.

- The sub-procedure $checkSAT(x)$ determines the satisfiability of an input formula x in a given SMT-LIB logic via an SMT solver.
- The sub-procedure $checkRealCE(P, t, CE)$ returns **true** if CE can be concretised to a counterexample[1], demonstrating a behaviour of P that falsifies t.
- The set $Lattices$ consists of all occurrences of lattices for all library functions in P.
- We denote by L_f^i a lattice for the i-th occurrence of f in P, and by e the current element in the lattice traversal. For an element e, we define $literals$ as the *conjunction of guarded literals* of e.
- The sub-procedure $traverse_{UNSAT}(Lattices)$ performs the traversal of the lattice from the current element e

[1]A counterexample (conjoined with the model) is tested by using a theory under which the library function is fully defined.

Algorithm 1: LB-CEGAR

```
Input  : Program P, safety property t, and set Lattices
Output: ⟨Safe⟩, ⟨Unsafe, CE⟩, or ⟨Unknown, ⊥⟩
φ ← P ∧ ¬t
Query ← φ
⟨result, CE⟩ ← checkSAT(Query)
if result is UNSAT or checkRealCE(φ, CE) then
  | go to  Exit // No lattice-based refinement needed
end
χ ← true
repeat
  | χ' ← χ // Formula from the previous iteration
  | if result is UNSAT then
  |   | traverse_UNSAT(Lattices)
  | end
  | if result is SAT then
  |   | traverse_SAT(Lattices)
  | end
  | χ ← literals(φ, Lattices)
  | // Solve again if there are new literals
  | if χ ≠ χ' then
  |   | Query ← φ ∧ χ
  |   | ⟨result, CE⟩ ← checkSAT(Query)
  | end
until (χ = χ') or checkRealCE(Query, CE) or
  termination(result, Lattices);
Exit: // End of LB-CEGAR
if result is UNSAT then
  | return ⟨Safe⟩ // Safe
end
if checkRealCE(P, t, CE) then
  | return ⟨Unsafe, CE⟩ // Real counterexample
end
return ⟨Unknown⟩ // Inconclusive, further refinement needed
```

to the next element e' if the result of model-checking $\varphi \wedge literals$ is **UNSAT**. The next element e' in the same lattice as e is a 'sibling' of e, that is, an element, whose set of literals corresponds to a different subdomain of the input. If there is already a frontier of elements in each lattice such that model-checking $\varphi \wedge literals$ returns **UNSAT** for each element of these frontiers, the procedure $traverse_{UNSAT}(Lattices)$ does not change the current element e.

- The sub-procedure $traverse_{SAT}(Lattices)$ is invoked when there is a satisfying assignment for $\varphi \wedge literals$, but the counterexample induced by the assignment is *spurious*, that is, it does not correspond to a behaviour of P falsifying t. The procedure traverses the lattice to an element e' that refines e, that is, $S(e) \subset S(e')$. If e is a maximal element, the procedure $traverse_{SAT}(Lattices)$ does not change the current element e.
- In both sub-procedures $traverse_{UNSAT}$ and $traverse_{SAT}$, the lattices are traversed either in an arbitrary order or in an order determined by heuristics. We describe such heuristics in Sec. V-C.
- The sub-procedure $termination(result, Lattices)$ checks whether one of the three termination conditions holds: (1) the current satisfying assignment induces a concrete counterexamples, (2) there is an **UNSAT** frontier for each lattice $L_f^i \in Lattices$, or (3) there is a satisfying assignment for each maximal element in each lattice in $Lattices$ that does not induce a concrete counterexample.

Finally, we address the complexity resulting from having several functions in P, whose lattices refer to each other. This is illustrated by Ex. 1, where the correctness of the program depends on the guarded literal

$$(assume(\mathbf{true})) \wedge (\sin^2 x + \cos^2 x) = 1.$$

In fact, this is quite common in programs with trigonometric functions, as trigonometric identities often refer to several functions in the same identity. The algorithm identifies library functions used in the set $Lattices$ and assigns the same variable to all occurrences of the same function, hence connecting between the lattices of different functions.

C. Correctness and complexity of LB-CEGAR

It is easy to see that LB-CEGAR terminates. The lattice traversal visits every combination of elements of lattices in $Lattices$ at most once, and for each combination of elements it invokes the model-checking procedure of a bounded loop-free program P with respect to t, which terminates assuming terminating SMT queries.

The number of possible combinations of elements in the lattices is exponential in the number of lattices, hence leading to the complexity result below.

Theorem 1: The worst-case time complexity of LB-CEGAR is $O(|L|^n \times MC(P,t))$, where $|L|$ is the bound on the size of each lattice in the set $Lattices$, n is the number of lattices in $Lattices$, and $MC(P,t)$ is the time complexity of model-checking P with respect to t using the guarded literals.

Moreover, the following theorem states that LB-CEGAR produces a correct result.

Theorem 2: The following holds for any bounded loop-free program P and a safety property t, assuming correctness of the guarded literals in $Lattices$:

- If LB-CEGAR outputs **Safe**, the program P is correct with respect to t.
- If LB-CEGAR outputs **Unsafe** with an accompanying CE, the CE demonstrates an execution of P that falsifies t.
- If LB-CEGAR outputs **Unknown**, the current theory and the set of guarded literals are insufficient to produce a conclusive result.

We observe that, while the worst-case complexity of LB-CEGAR is exponential in the number of lattices, in practice the algorithm is very efficient, as we show in Sec. VI. This is partly due to the *incrementality* of the calls to the SMT solver, as the formula φ representing $P \wedge \neg t$ stays the same for all iterations, and the next element e' differs from the current element e of the lattice only slightly. Another reason for the significantly lower complexity in practice is that our implementation of LB-CEGAR includes several heuristics, which we describe in the next section. The heuristics do not alter the correctness of the algorithm.

V. Implementation

The algorithms were implemented on top of the SMT-based function summarisation bounded model checker HiFrog [15] with OpenSMT [22] and Z3 [7], [23] solvers. The details of our initial implementation are described in [18]. Here we describe the extension of the implementation to support the full LB-CEGAR algorithm.

Fig. 2 presents a high-level view of the implementation of LB-CEGAR in HiFrog and a comparison between the implementation as a flat (non-hierarchical) set of user-defined summaries, our prototype implementation with one occurrence of one function, and the current implementation of the general algorithm.

A. Pre-processing stage

We constructed two lattices for $\sin$ and $\cos$ functions via a set of BASH scripts (see [18]) for the evaluation of our approach. The guarded literals were imported from the raw data of Coq proof assistant [30] and Wikipedia [31], [32] and translated to SMT summaries. The definitions of constants (e.g., π from $math.h$) and trigonometric tables (values of the trigonometric functions for $x = c \cdot \pi$, for some $c \in \mathbb{N}$) were added to the set of guarded literals manually. The final set consists of 80 guarded literals and was used to construct the meet-semilattices for $\sin x$ and $\cos x$ functions. Textual files of these meet-semilattices are available at [24], [26].

B. Implementation in HiFrog

The implementation of LB-CEGAR uploads only the set of guarded literals in the current element of the lattice. If the current element is insufficient for solving the formula (that is, the satisfying assignments produced by the SMT solver do not induce concrete counterexamples), the algorithm traverses the lattice to a higher element, translated in the implementation to adding and removing some subsets of guarded literals. It is clear that the new formula only differs from the one in the previous iteration by a subset of guarded literals. The implementation exploits this fact by using the SMT solver in an *incremental* mode.

We extended the support for incremental solving in HiFrog, adding non-, semi-, and full-incremental solving modes, to support different degrees of incrementality (e.g., semi-incremental solving mode allows only $push()$ calls). With this support, the implementation only modifies a single query from one iteration to the next, which is less costly than re-writing the whole formula.

C. Heuristics

We implemented the following heuristics to improve the complexity of lattice traversal in LB-CEGAR. While none of these heuristics change the worst-case time complexity, our experiments show that they are beneficial on programs in our benchmark set.

- The choice of the successor in the sub-procedure $traverse_{SAT}(Lattices)$ is done based on the current spurious counterexample CE, similarly to the traditional CEGAR. We identify the location in the code where the abstract counterexample deviates from a concrete execution and use this information to guide the lattice traversal to the element that refines this particular location (if such an element exists).
- The 'frontier of unsatisfiability', that is, a frontier of a lattice that results in **UNSAT** for each element of this frontier, is computed once per lattice and is fixed. While in theory it is possible that the current frontier of a lattice L_1 results in **UNSAT** when combined with an element e of a lattice L_2, but not with an element e' of L_2, in practice such cases are rare. There is an option to output **Unknown** if the set of frontiers computed gradually does not result in **UNSAT**, thus potentially increasing the number of cases, where LB-CEGAR outputs an inconclusive result. In our experiments, this heuristic does not lead to an increase in the number of inconclusive results.
- For lattices representing different occurrences of a function f in P which occur in a loop, we traverse these lattices simultaneously. The motivation for the 'coordinated' traversal is that all loop iterations, except, perhaps, for the last one, are similar, and hence there is a high probability that the same set of guarded literals would fit all these occurrences.

VI. Evaluation

For the evaluation of LB-CEGAR, we constructed two lattices for $\sin$ and $\cos$ functions with 40 and 38 guarded literals, respectively. The validation test for these expressions contains a set of 144 benchmarks in C with a total of 365 assert statements. The scripts for the lattice construction, the benchmarks for the validation test, and the results of the validation test are available at [24], [25].

The set of benchmarks contained a mix of our crafted benchmarks, programs from the software verification competition SVCOMP [33], and HiFrog benchmarks [15], with a total of 141 C programs with at least one library function call, containing in total 194 calls for $\sin$ and 179 calls for $\cos$, with 279 claims (127 **SAT** and 152 **UNSAT**). In 42 benchmarks, the library function is called at least 4 times, and in 8 benchmarks, the library function call is in a loop. The crafted benchmarks either assert known properties of trigonometric functions or contain a small part of code that is typical to kinematic problems, mainly examining the ability of verifying code with multiplication between two library function calls; e.g., $\cos\phi \cdot \sin\theta$.

To model a program with its property, we either used the quantifier-free SMT theory for equality logic with uninterpreted functions (EUF) with a semi-incremental solving mode in the OpenSMT solver [22] or used the quantifier-free SMT theories for linear arithmetics (LA) with EUF with an incremental solving mode in the Z3 solver [23]. For the CEGAR-style check of counterexamples in the sub-procedure $checkRealCE$ in Sec. IV-B, we used the quantifier-free SMT theory for non-linear real arithmetic (NRA).

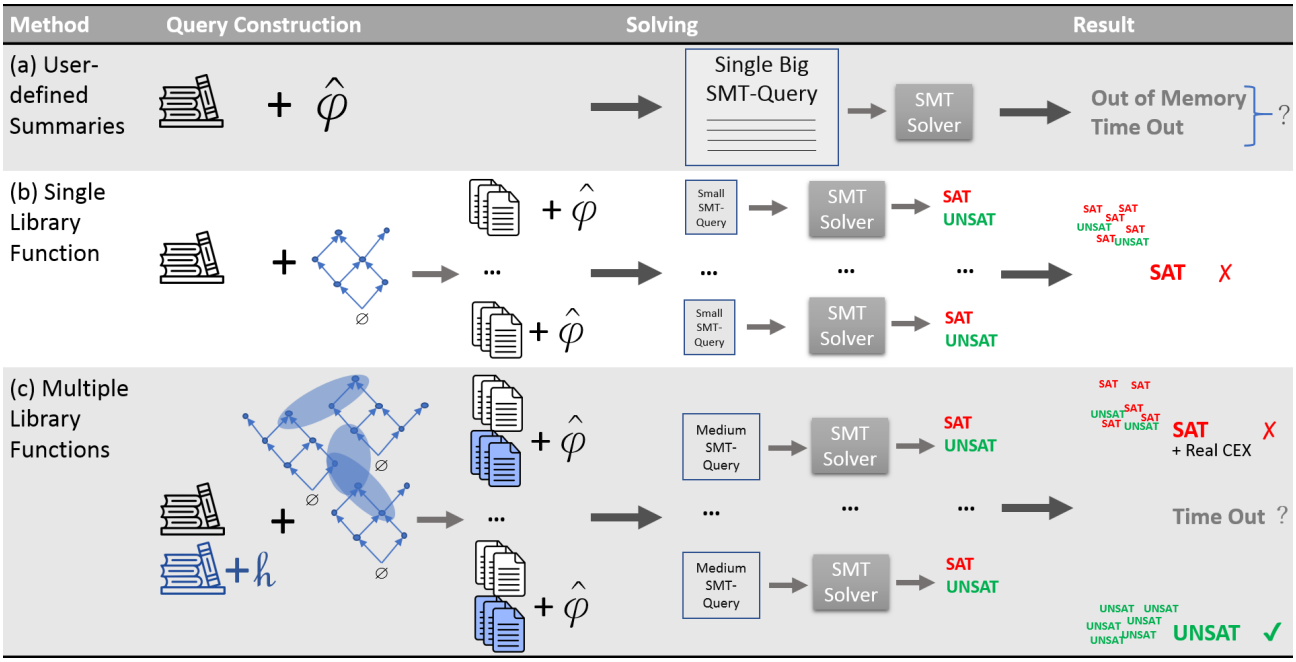

Figure 2. *LB-CEGAR for a program P with several library functions (c) in comparison with BMC with user-defined summaries (a) and BMC with the lattice-based refinement initial approach (b).*

The default parameters of the LB-CEGAR algorithm include the use of all the heuristics in Sec. V-C. In the evaluation, we used the default parameters and thus used all of the three heuristics; see [24] for more details regarding these parameters.

The experiments were performed on a virtual machine (VM) with Ubuntu 16.04 Linux system, single-core, 8GB RAM; the VM runs on a machine with 4-Intel i7-6600U CPUs clocked at 2.60GHz. The experimental results, the benchmarks, and the source code, are available at [24], [26].

A. Evaluation of LB-CEGAR with Real arithmetics

Figure 3 presents the comparison of LB-CEGAR with CBMC version 5.10 [34], HiFrog [15], and our previous implementation [18] supporting one library function at a time.

The total number of **solved instances** is the blue bar and the orange bar, for **Safe** and **Unsafe** instances respectively. The total number (as a negative number) of **unsolved instances** is the gray bar and the yellow bar, for **SAT** instances that are classified as **Unknown** (or **SAT** without a counterexample), and for the instances that timed out (TO) or were out-of-memory (OM), with the time-out set to 4000s and out-of-memory set to 3GB, respectively.

The four different colours of the bars are consistent across all six charts. Each chart represents the total solved instances for a particular tool or a variant of a tool. The tools at the clockwise order are, LB-CEGAR (top-left), CBMC [35] (top-right), HiFrog [15] LRA (middle-right), EUF (bottom-right) and with user defined summaries (bottom-left), and HiFrog [18] with a single lattice (middle-left). Note that, HiFrog used either EUF **or** LRA (the quantifier-free SMT theory for linear real arithmetic). All approaches with summaries used EUF with LRA (UFLRA).

Verification with function summaries in HiFrog avoids processing the single static assignment (SSA) expression of the original function, and only uses SMT summaries which for the experimental section here[2], contained no computation of the actual function nor of its Taylor series approximations, at any stage. Hence, complicated benchmarks, which contain library function calls in a loop or a non-linear expression, were more likely to be successfully verified with summary-based approaches than with classical approaches which usually require computation (up to some precision) of an approximation of the function and ran out of resources eventually.

HiFrog with user-defined summaries used ~ 80 equations of trigonometric properties, loaded at once as unstructured data and solved a total of 227 instances (HiFrog - User-Defined, bottom-left, Fig. 3). HiFrog with a single lattice used ~ 40 equations of trigonometric properties, and solved a total of 185 instances (Single-Lattice - LRA, middle-left, Fig. 3).

LB-CEGAR with two lattices, constructed from a set of ~ 80 equations of trigonometric properties (LB-CEGAR graph, top-left, Fig. 3), outperformed both variants of HiFrog and had the highest overall number of *solved instances*: over **260** instances.

Connecting lattices of different functions in LB-CEGAR algorithm allowed our approach to verify the highest number of **Safe** instances, on many on which other tools failed (including HiFrog-User-Defined and Single-Lattice-LRA). In

[2]There is no limitation on using a summary with the actual definition of a function with our method in general, as we have shown in [18] for Modulo function. However, the generalised technique in this paper designed to perform well, even without such a summary, as schematically shown in Fig. 2 (c).

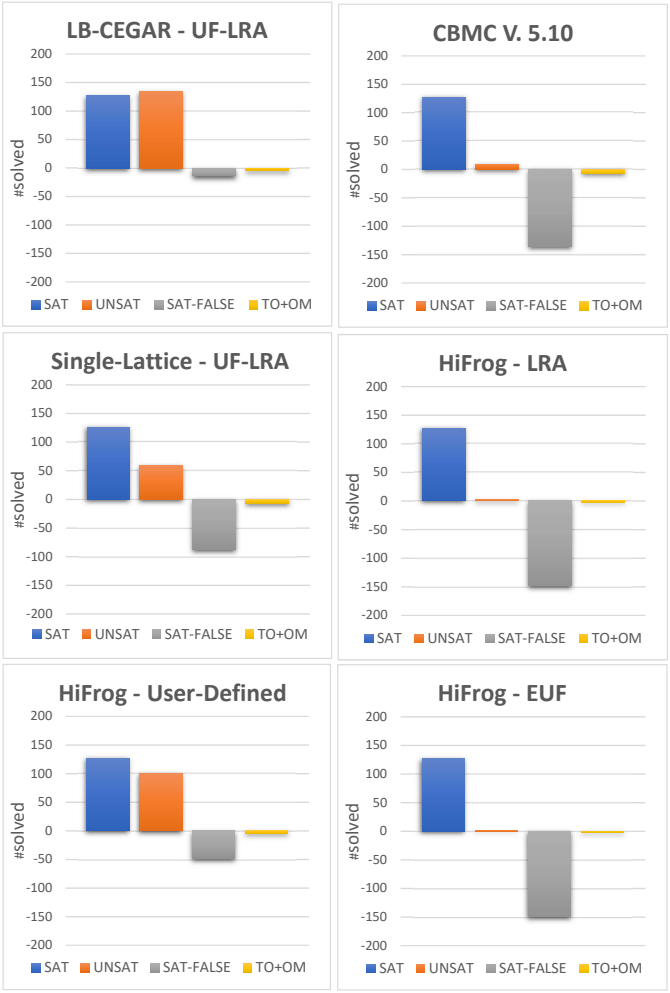

Figure 3. *Comparison of the Number (#) of Solved Claims with Different Approaches and a Set of Trigonometric Benchmarks in C.*

comparison with LB-CEGAR (with 261 solved instances, out of 279), other tools that do not use summaries or insights on trigonometric functions, managed to solve at most 136 instances (CBMC V. 5. 10, HiFrog-LRA, and HiFrog-EUF, in Fig. 3, at top-right, bottom-middle, and bottom-right, respectively), mainly because of the use of non-deterministic variables to represent trigonometric functions.

B. Evaluation of LB-CEGAR with different parameters of HiFrog

Table I presents a full comparison of our implementation of the LB-CEGAR algorithm with other tools. The comparison was performed using various parameters of HiFrog, even-though LRA with EUF is the most suitable theory-combination for trigonometric functions, based on our experience.

The physical files of SMT summaries for LRA and EUF (UFLRA) were the same; HiFrog read an SMT summary file differently according to the theory in use. The SMT summaries for the quantifier-free SMT theory for linear integer arithmetic (LIA) were different to prevent any conversions to real arithmetic (e.g., to_real token) in the SMT query where all guards and literals were modelled via LIA with EUF (UFLIA).

In Table I, the verification results of LB-CEGAR appear in white and are compared to CBMC [35] and HiFrog [15], [18] (grey and dark grey), with the best results underlined and marked in light-yellow.

The evaluation of HiFrog in grey scale is with a single lattice, and with and without user defined summaries, using: EUF, LRA with EUF (UFLRA), LIA with EUF (UFLIA), each of which is a different column in Table I (with the theory being EUF, LRA or LIA). The symbol # stands for the number of instances solved (first two lines) or unsolved (last two lines). Unsolved instances are false-negative results (FN-SAT), inconclusive (also marked as FN-SAT), or timeout or out-of-memory ($TO + OM$).

The description of each column in Table I is as follows.

- The **white** columns in Table I (3 columns, **LB-CEGAR col.**) contain the results the LB-CEGAR algorithm along with the modifications presented in Sec. IV with different sets of parameters (with the theory being EUF, LRA, or LIA), against other tools in the gray scale columns.
- The **gray scale** columns in Table I (10 columns from the end) contain the results of other tools: the lattice-based refinement approach with a single lattice [18] in **Lattice Single col.**, HiFrog with a large set of user defined summaries [15] in **HiFrog UDS col.**, and HiFrog [15] and CBMC version 5.10 [34] in the **most right columns**.

The variant of HiFrog with theory refinement is omitted from the comparison, as its current implementation does not support trigonometric functions.

For the lattice-based refinement approach for verification of programs with multiple trigonometric functions, the best setting was LRA with EUF (LB-CEGAR, LRA col.), which had the highest overall number of *solved instances* over **260** instances, and performed almost as well as HiFrog without any summary (ran out of resources 4 times vs. 2 times HiFrog did). The other two configurations of parameters that we tried with the lattice-based refinement approach for programs with multiple library functions, were EUF (LB-CEGAR, EUF col.) and LIA with EUF (LB-CEGAR, LIA col.). While EUF performed poorly in general, LIA with EUF has shown limited potential in solving instances that required real arithmetics, which indicates the possibility of applying this method for code with library functions over significantly different input domains, that is, code with both continuous and discontinuous functions.

The comparison with a single lattice [18] (Lattice Single, LRA col.) used either a meet-semilattice for $\sin$ function or for $\cos$ function per benchmark, which led to poor performance when both lattices were required; however, perhaps unsurprisingly, this did not result in poor performance when a single lattice was sufficient to prove safety of a claim (59 instances, Lattice Single, LRA col.). HiFrog with user-defined summaries (HiFrog UDS, LRA col.) could not solve around 50 safe instances that required a wider context regarding other

Table I
COMPARISON OF LB-CEGAR IN HIFROG WITH DIFFERENT PARAMETERS FOR CBMC AND HIFROG. WE REPORT THE NUMBER OF SOLVED AND UNSOLVED INSTANCES, WITH THE TIME-OUT (TO) SET TO 4,000S AND MEMORY LIMITED TO 3GB (OM).

		LB-CEGAR			**Lattice Single**			**HiFrog UDS**			**HiFrog**			**CBMC**
		LRA	LIA	EUF	LRA	LIA	EUF	LRA	LIA	EUF	LRA	LIA	EUF	
#Solved	UNSAT	**134**	42	8	59	13	6	100	24	9	2	3	1	9
	SAT	**127**	126	126	126	125	126	**127**	126	126	**127**	**127**	**127**	**127**
#Failed	FN	**14**	104	136	87	133	139	48	122	136	148	147	149	136
	TO+OM	4	7	9	7	8	8	4	7	8	**2**	**2**	**2**	7

library functions, for one or more expressions with a library function call.

VII. RELATED WORK

The problem of verification of programs with transcendental functions and, in particular, trigonometric functions is addressed by several verification tools, such as iSAT3 [36] via interval propagation by refining the computed interval bounds, dReal [37] by also using interval propagation and δ-satisfiability but with user-specified precision where δ is associated with the numerical error, Coq interval [38] and Gappa [39] via interval propagation with Taylor series, and MathSAT5 [6], [12], [40] by using Taylor series with a partial set of trigonometric properties and for hardware verification [41]. In contrast to these approaches, our algorithm does not require nonlinear arithmetic or a calculation of Taylor series, which is computationally expensive for large programs.

Computationally inexpensive theories can be used to over-approximate complex problems. This approach has been used in solving equations on non-linear real arithmetic and transcendental functions based on linear real arithmetic and equality logic with uninterpreted functions [6], [12], [42], [43], as well as on scaling up bit-vector solving [15], [44], [45]. Our work can be seen as a generalisation of these approaches as we support inclusion of lemmas from more descriptive logics to increase the expressiveness of computationally lighter logics.

Lattices are a useful mathematical structure in understanding the relationships between different abstractions and have been widely applied in program solving with Craig interpolation. [46] presents a semantic solver-independent framework for systematically exploring interpolant lattices using the notion of interpolation abstraction. A lattice-based system for interpolation in propositional structures is presented in [47], extended to consider size optimisation techniques in the context of function summaries in [48], [49], and further extended to partial variable assignments in [50]. Similar lattice-based reasoning has also been extended to interpolation in first order logic with other SMT theories [51], [52]. The approach presented in this work differs from the above in that we do not rely on interpolation and work in tight integration with the model-checker.

Lattices and posets are used in abstract interpretation [53] to model a sound approximation of the semantics of code, where completeness and partial completeness [54]–[57] refer to the absence of loss of precision during the approximation of the semantics of code. Giacobazzi et al. [56], [57] present the notation of backward and forward completeness and show the connection between iteratively computing the backward (forward)-complete shell to the general CEGAR framework [17]. The completeness of their algorithm depends on the properties of the abstraction, while our algorithm has no such requirements.

The idea of applying SMT solvers with abstract interpretation for program verification to deal with program properties that are often difficult to model was presented in [58] and generalised into an efficient framework with fix-point completeness in [59], [60]. For trigonometric functions where the implementation is usually based on Taylor approximations, the program properties and the properties of trigonometric functions in mathematics differ substantially.

VIII. CONCLUSIONS AND FUTURE WORK

We presented a new algorithm LB-CEGAR that is used for verification of programs with library functions, for which a number of equations, some of which are instrumental for verification of these programs, exist in external sources (the mathematical library, Coq proof assistant, etc.). The main idea of the algorithm is to organize the equations in subset lattices, and to replace the traditional CEGAR refinement loop with lattice traversal. The algorithm is general in the sense that it allows several occurrences of the same library function and/or several different library functions, some of which depend on each other, in the same program. While the theoretical worst-case complexity of LB-CEGAR is high due to an exponential number of combinations of elements of different lattices, our experimental results show that the algorithm is very efficient in practice and outperforms state-of-the-art model-checking tools on benchmarks with trigonometric functions.

We view the programs with trigonometric functions as the primary domain of application of LB-CEGAR. In the future, we plan to explore the domain of verification of programs describing robots' movements and kinematics in general.

ACKNOWLEDGEMENTS.

This work was partially supported by the Swiss National Science Foundation (SNSF) grant 200020_166288.

References

[1] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking," *J. ACM*, vol. 52, no. 3, pp. 365–473, 2005.

[2] "ROS homepage," Date Accessed May 01, 2019. [Online]. Available: http://www.ros.org/

[3] J. Wittenburg, *Kinematics: Theory and Applications*. Springer, 2016.

[4] "Open Source Physics page," Date Accessed May 01, 2019. [Online]. Available: http://www.compadre.org/osp/

[5] B. Akbarpour and L. C. Paulson, "Metitarski: An automatic theorem prover for real-valued special functions," *J. Automat. Reason*, vol. 44, no. 3, pp. 175–205, 2010.

[6] A. Cimatti, A. Griggio, A. Irfan, M. Roveri, and R. Sebastiani, "Satisfiability modulo transcendental functions via incremental linearization," in *CADE*, ser. LNCS, vol. 10395. Springer, 2017, pp. 95–113.

[7] L. De Moura and G. O. Passmore, "Computation in real closed infinitesimal and transcendental extensions of the rationals," in *CADE*, ser. LNCS, vol. 7898. Springer, 2013, pp. 178–192.

[8] W. Denman and C. Muñoz, "Automated real proving in pvs via metitarski," in *FM*, ser. LNCS, vol. 8442. Springer, 2014, pp. 194–199.

[9] N. Ge, E. Jenn, N. Breton, and Y. Fonteneau, "Integrated formal verification of safety-critical software," *Int J. Softw Tools Technol Transf.*, vol. 20, no. 4, pp. 423–440, 2018.

[10] P. Trojanek and K. Eder, "Verification and testing of mobile robot navigation algorithms: A case study in spark," in *IROS*. IEEE, 2014, pp. 1489–1494.

[11] R. Kirner, M. Grössing, and P. P. Puschner, "Comparing WCET and resource demands of trigonometric functions implemented as iterative calculations vs. table-lookup," in *WCET*, ser. OASICS. IBFI, Schloss Dagstuhl, Germany, 2006.

[12] A. Cimatti, A. Griggio, A. Irfan, M. Roveri, and R. Sebastiani, "Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions," *ACM Trans. Comput. Logic*, vol. 19, no. 3, pp. 19:1–19:52, 2018.

[13] M. Daumas, D. Lester, and C. Muñoz, "Verified real number calculations: A library for interval arithmetic," *IEEE Trans. Comput.*, vol. 58, no. 2, pp. 226–237, 2009.

[14] G. Melquiond and C. Munoz, "Guaranteed proofs using interval arithmetic," in *ARITH*. IEEE, 2005, pp. 188–195.

[15] L. Alt, S. Asadi, H. Chockler, K. Even Mendoza, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, "HiFrog: SMT-based function summarization for software verification," in *TACAS*, ser. LNCS, vol. 10206. Springer, 2017, pp. 207–213.

[16] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *CAV*, ser. LNCS, vol. 1855. Springer, 2000, pp. 154–169.

[17] ——, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[18] K. Even-Mendoza, S. Asadi, A. E. J. Hyvärinen, H. Chockler, and N. Sharygina, "Lattice-based refinement in bounded model checking," in *VSTTE*, ser. LNCS, vol. 11294. Springer, 2018, pp. 50–68.

[19] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using smt solvers instead of sat solvers," in *SPIN*, ser. LNCS, vol. 3925. Springer, 2006, pp. 146–162.

[20] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf, "Jayhorn: A framework for verifying java programs," in *CAV*, ser. LNCS, vol. 9779. Springer, 2016, pp. 352–358.

[21] D. Brizhinev and R. Goré, "A case study in formal verification of a java program," *arXiv preprint arXiv:1809.03162*, 2018.

[22] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina, "OpenSMT2: An SMT solver for multi-core and cloud computing," in *SAT*, ser. LNCS, vol. 9710. Springer, 2016, pp. 547–553.

[23] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[24] "LB-CEGAR page," 2019. [Online]. Available: http://verify.inf.usi.ch/content/lattice-refinement/

[25] "Git repository: LB-CEGAR evaluation and additional info." 2019. [Online]. Available: https://github.com/karineek/latticeref/

[26] "Git repository of HiFrog," Date Accessed May 07, 2019. [Online]. Available: https://scm.ti-edu.ch/projects/hifrog/

[27] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," in *J. Symb. Log.*, 1957, pp. 269–285.

[28] "HiFrog - tool usage page," 2017. [Online]. Available: http://verify.inf.usi.ch/hifrog/tool-usage/

[29] G. Birkhoff, *Lattice Theory*, 3rd ed. AMS, 1967.

[30] coq.inria.fr, "The Coq proof assistant," Date Accessed May 01, 2019. [Online]. Available: https://coq.inria.fr/

[31] "List of trigonometric identities, from Wikipedia, the free encyclopedia," Date Accessed May 01, 2019. [Online]. Available: https://en.wikipedia.org/wiki/List_of_trigonometric_identities/

[32] "Trigonometric tables, from Wikipedia, the free encyclopedia," Date Accessed May 01, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Trigonometric_tables/

[33] 2018. [Online]. Available: https://sv-comp.sosy-lab.org/2018/

[34] Date Accessed May 01, 2019. [Online]. Available: http://www.cprover.org/cbmc/

[35] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.

[36] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert, "Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure," *J. Satisf. Boolean Model. Comput.*, vol. 1, pp. 209–236, 2007.

[37] S. Gao, S. Kong, and E. M. Clarke, "dreal: An smt solver for nonlinear theories over the reals," in *CADE*, ser. LNAI, vol. 7898. Springer, 2013, pp. 208–214.

[38] G. Melquiond, "Floating-point arithmetic in the coq system," *Inf. Comput.*, vol. 216, pp. 14 – 23, 2012, Special Issue: 8th Conference on Real Numbers and Computers.

[39] F. de Dinechin, C. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using gappa," *IEEE Trans. Comput.*, vol. 60, no. 2, pp. 242–253, 2011.

[40] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *TACAS*, ser. LNCS, vol. 7795. Springer, 2013, pp. 93–107.

[41] J. Harrison, "Formal verification of floating point trigonometric functions," in *FMCAD*, ser. LNCS, vol. 1954. Springer, 2000, pp. 217–233.

[42] A. Cimatti, A. Griggio, A. Irfan, M. Roveri, and R. Sebastiani, "Invariant checking of NRA transition systems via incremental reduction to LRA with EUF," in *TACAS*, ser. LNCS, vol. 10205. Springer, 2017, pp. 58–75.

[43] T. Kutsuna, Y. Ishii, and A. Yamamoto, "Abstraction and refinement of mathematical functions toward smt-based test-case generation," *Int J. Softw Tools Technol Transf.*, vol. 18, no. 1, pp. 109–120, 2016.

[44] Y.-S. Ho, P. Chauhan, P. Roy, A. Mishchenko, and R. Brayton, "Efficient uninterpreted function abstraction and refinement for word-level model checking," in *FMCAD*. ACM, 2016, pp. 65–72.

[45] A. E. J. Hyvärinen, S. Asadi, K. Even-Mendoza, G. Fedyukovich, H. Chockler, and N. Sharygina, "Theory refinement for program verification," in *SAT*, ser. LNCS, vol. 10491. Springer, 2017, pp. 347–363.

[46] P. Rummer and P. Subotic, "Exploring interpolants," in *FMCAD*. IEEE, 2013, pp. 69–76.

[47] V. D'Silva, M. Purandare, G. Weissenbacher, and D. Kroening, "Interpolant strength," in *VMCAI*, ser. LNCS, vol. 5944. Springer, 2010, pp. 129–145.

[48] L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, "A Proof-Sensitive Approach for Small Propositional Interpolants," in *VSTTE*, ser. LNCS, vol. 9593. Springer, 2015, pp. 1–18.

[49] S. F. Rollini, L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, "PeRIPLO: A framework for producing effective interpolants in SAT-based software verification," in *LPAR*, ser. LNCS, vol. 8312. Springer, 2013, pp. 683–693.

[50] P. Jancík, L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, J. Kofron, and N. Sharygina, "PVAIR: Partial Variable Assignment InterpolatoR," in *FASE*, ser. LNCS, vol. 9633. Springer, 2016, pp. 419–434.

[51] L. Alt, A. E. J. Hyvärinen, and N. Sharygina, "LRA interpolants from no man's land," in *HVC*, ser. LNCS, vol. 10629. Springer, 2017, pp. 195–210.

[52] L. Alt, A. E. J. Hyvärinen, S. Asadi, and N. Sharygina, "Duality-based interpolation for quantifier-free equalities and uninterpreted functions," in *FMCAD*. IEEE, 2017, pp. 39–46.

[53] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM, 1977, pp. 238–252.

[54] P. Cousot, "Partial completeness of abstract fixpoint checking," in *SARA*, ser. LNAI, vol. 1864. Springer, 2000, pp. 1–25.

[55] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *POPL*. ACM, 1979, pp. 269–282.

[56] R. Giacobazzi and E. Quintarelli, "Incompleteness, counterexamples, and refinements in abstract model-checking," in *SAS*, ser. LNCS, vol. 2126. Springer, 2001, pp. 356–373.
[57] R. Giacobazzi, F. Ranzato, and F. Scozzari, "Making abstract interpretations complete," *J. ACM*, vol. 47, no. 2, pp. 361–416, 2000.
[58] P. Cousot, R. Cousot, and L. Mauborgne, "Theories, solvers and static analysis by abstract interpretation," *J. ACM*, vol. 59, no. 6, p. 31, 2012.
[59] V. D'Silva, L. Haller, and D. Kroening, "Abstract satisfaction," in *POPL*. ACM, 2014, pp. 139–150.
[60] L. Haller, "Abstract satisfaction," Ph.D. dissertation, Uni. Oxford, 2013.