Lattice-Based Refinement in Bounded Model Checking

Karine Even-Mendoza[1], Sepideh Asadi[2], Antti E. J. Hyvärinen[2], Hana Chockler[1], and Natasha Sharygina[2]

[1] King's College London, UK
karine.even_mendoza@kcl.ac.uk, hana.chockler@kcl.ac.uk
[2] Università della Svizzera italiana, Switzerland
antti.hyvaerinen@usi.ch, sepideh.asadi@usi.ch, natasha.sharygina@usi.ch

Abstract. In this paper we present an algorithm for bounded model-checking with SMT solvers of programs with library functions — either standard or user-defined. Typically, if the program correctness depends on the output of a library function, the model-checking process either treats this function as an uninterpreted function, or is required to use a theory under which the function in question is fully defined. The former approach leads to numerous spurious counter-examples, whereas the later faces the danger of the state-explosion problem, where the resulting formula is too large to be solved by means of modern SMT solvers.

We extend the approach of user-defined summaries and propose to represent the set of existing summaries for a given library function as a *lattice* of subsets of summaries, with the meet and join operations defined as intersection and union, respectively. The refinement process is then triggered by the lattice traversal, where in each node the SMT solver uses the subset of SMT summaries stored in this node to search for a satisfying assignment. The direction of the traversal is determined by the results of the concretisation of an abstract counterexample obtained at the current node. Our experimental results demonstrate that this approach allows to solve a number of instances that were previously unsolvable by the existing bounded model-checkers.

1 Introduction

Bounded model checking (BMC) amounts to verifying correctness of a given program within the given bound on the maximal number of loop iterations and recursion depth [10]. It has been shown very effective in finding errors in programs, as many errors manifest themselves in short executions. As the programs usually induce a very large state space even at bounded depth, there is a need for scalable tools to make the verification process efficient. The satisfiability modulo theories (SMT) [22] reasoning framework is currently one of the most successful approaches to verifying software in a scalable way. The approach is based on modeling 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 behavior relevant to correctness, while sufficiently high-level to prevent reasoning from becoming prohibitively expensive — the process known as *theory refinement* [28]. Since in general more precise theories are more expensive computationally, finding such a balance is a non-trivial task. Moreover, often there is no need to refine the theory for the whole program. As the modern approach to software development encourages modular development and re-use of components, programs increasingly use library functions, defined elsewhere. If the correctness of the program depends on the implementation of the library (or user-defined) functions, there is a need for a modular approach that allows us to refine only the relevant functions. Yet, currently, the theory refinement is not performed on the granularity level of a single function, hence BMC of even simple programs can result in a state explosion, especially if the library function is called inside a loop.

In this paper, we introduce an approach to efficient SMT-based bounded model checking with lattices of summaries for library functions, either taken from known properties of the functions or user-defined. Roughly speaking, the lattice is a *subset lattice*, where each element represents a subset of Boolean expressions (that we call *facts*) that hold for some subset of inputs to the function; the *join* and *meet* operators are defined as union and intersection, respectively (see Sec. 2 for the formal definition). The counter-example-guided abstraction refinement (CEGAR) [14,16] that we describe in this paper is lattice-based, is triggered by a traversal of the lattice, and the CEGAR loop is repeated until one of the following outcomes occurs: (i) we prove correctness of the bounded program (that is, absence of concrete counterexamples), (ii) we find a concrete counterexample, or (iii) the current theory together with the equalities in the lattice is determined insufficient for reaching a conclusion.

The following motivational example illustrates the use of lattices with LRA (quantifier-free linear real arithmetics) theory.

Example 1. The code example in Fig. 1 describes the *greatest common divisor* (GCD) algorithm. We assume that both inputs are positive integers. The program is safe with respect to the assertion $g \leq x$. However, with the LRA theory, an SMT solver cannot prove correctness of the program, as GCD is not expressible in LRA. The standard approach is to have $gcd(x, y)$ assume any real value; thus, attempting to verify this program with an SMT solver and the LRA theory results in an infinite number of spurious counterexamples. In the example, we augment the solver with a set of *facts* about the *modulo* function, arranged in a meet semilattice. These facts are taken from an existing set of lemmas and theorems of the Coq proof assistant [3] for $a\%n$:

$$f_1 \equiv z\_mod\_mult \equiv$$

$$\equiv a \ \ mod \ \ n = 0 \text{ with the assumption } a == x * n \text{ for some positive integer } x;$$

$$f_2 \equiv z\_mod\_pos\_bound \wedge z\_mod\_unique \equiv$$

$$\equiv (0 \leq a \ \ mod \ \ n < n) \wedge (0 \leq r < n \implies a = n * q + r \implies r = a \ \ mod \ \ n)$$

$$\text{for some positive integers } r \text{ and } q, \text{ with the assumption } (n > 0) \wedge (a \neq x * n);$$

$$f_3 \equiv z\_mod\_remainder \wedge z\_mod\_unique\_full \equiv$$

```c
int gcd(int x, int y)
{
   int tmp;
   while(y != 0) {
      tmp = x%y;
      x=y;
      y=tmp; }
   return x;
}
```

```c
int main(void)
{
   int x=45;
   int y=18;
   int g = gcd(x,y);

   assert(g <= x);
}
```

Fig. 1. *The GCD program using modulo function.*

$$\equiv (n \neq 0 \implies (0 \leq a \ \ mod \ \ n < n \vee n < a \ \ mod \ \ n <= 0)) \wedge ((0 \leq r < n \vee n < r \leq 0) \\ \implies a = b * q + r \implies r = a \ \ mod \ \ n) \text{ with the assumption } \textbf{true}.$$

The assumptions are different from the original guards in [3], as these are rewritten during the build of the meet semilattice. The original subset lattice consists of all subsets of the set $\{f_1, f_2, f_3\}$. It is analysed and reduced as described in Sec. 3 to remove contradicting facts and equivalent elements. In this example, the set $\{f_3\}$ generalises $\{f_1\} \sqcup \{f_2\}$. Fig. 2 shows the original subset lattice on the left, and the resulting meet semilattice of facts on the right. In the lattice

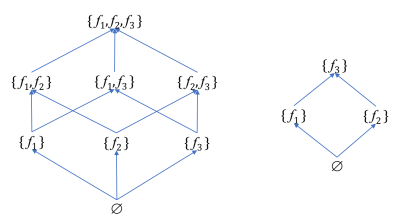

Fig. 2. Original subset lattice of facts and reduced meet semilattice for the *modulo* function in LRA.

traversal, we start from the bottom element $\emptyset$ and traverse the meet semilattice until we either prove that the program is safe or find a real counterexample (or show that a further theory refinement is needed). In this example, we traverse the lattice until the element $\{f_3\}$, which is sufficient to prove that the program is safe. Specifically, the fact f_1 is used to prove loop termination, and the fact f_2 is used to prove the assert.

Our algorithms are implemented in the bounded model checker HiFrog [5] supporting a subset of the C language and using the SMT solver OpenSMT [29]. We demonstrate the lattice construction on several examples of lattices for the *modulo* function. The facts for the lattice construction are obtained from the built-in theorems and statements in the Coq proof assistant [3].

Our preliminary experimental results show that lattice-traversal-based CEGAR can avoid the state-explosion problem and successfully solve programs that are not solvable using the standard CEGAR approach. The lattices are constructed using data from an independent source, and we show that even with a relatively small lattice we can verify benchmarks which either are impossible to verify in less precise theories or are too expensive to verify with the precise definition. Our set of benchmarks is a mix of our own crafted benchmarks and benchmarks from the software verification competition SVComp 2017 [4].

The full paper, HiFrog tool, and lattices and programs used in our experiments, are available at `http://verify.inf.usi.ch/content/lattice-refinement`.

Related Work. Lattices are useful in understanding the relationships between abstractions, and have been widely applied in particular in Craig interpolation [20]. For instance [33] presents a semantic and 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 [23], further extended in [32,6] to consider size optimisation techniques in the context of function summaries, and to partial variable assignments in [30]. Similar lattice-based reasoning has also been extended to interpolation in different SMT theories, including the equality logic with uninterpreted functions [8], and linear real arithmetics [7]. The approach presented in this work is different from these in that we do not rely on interpolation, and work in tight integration with model checking.

In addition to interpolation, also computationally inexpensive theories can be used to over-approximate complex problem. This approach has been used in solving equations on non-linear real arithmetics and transcendental functions based on linear real arithmetics and equality logic with uninterpreted functions [31,12,13]; as well as on scaling up bit-vector solving [27,5,28]. Parts of our work can be seen as a generalisation of such approaches as we support inclusion of lemmas from more descriptive logics to increase the expressiveness of computationally lighter logics.

Abstract interpretation [18] uses posets and lattices to model a sound approximation of the semantics of code. Partial completeness and completeness in abstract interpretation [17,19,25,26] refers to the no loss of precision during the approximation of the semantics of code. Giacobazzi et al. [25,26] present the notation of backward and forward completeness in abstract interpretation and show the connection between iteratively computing the backward (forward)-complete shell to the general CEGAR framework[16]; however the completeness of their algorithm depends on the properties of the abstraction while our algorithm has no such requirements.

Interesting work on combining theorem provers with SMT solvers include the SMTCoq system [24]. Our work uses facts from the Coq library, but differs from SMTCoq in that we import the facts directly to the SMT solver instead of giving the SMT solver to Coq.

2 Preliminaries

Lattices and subset lattices. For a given set X, the family of all subsets of X, partially ordered by the inclusion operator, forms a *subset lattice* $L(X)$. The $\sqcap$ and $\sqcup$ operators are defined on $L(X)$ as *intersection* and *union*, respectively. The top element $\top$ is the whole set X, and the bottom element $\bot$ is the empty set $\emptyset$. The height of the subset lattice $L(X)$ is $|X| + 1$, and all maximal chains have exactly $|X|+1$ elements. We note that $L(X)$ is a De-Morgan lattice [11], as meet and join distribute over each other. In this paper, we consider only lattices where X is a finite set.

A *meet-semilattice* is a partially ordered set that has a $\sqcap$ for any subset of its elements (but not necessarily $\sqcup$).

Bounded model checking. Let P be a loop-free program represented as a transition system, and a *safety property* t, that is, a logical formula over the variables of P. We are interested in determining whether all reachable states of P satisfy t. Given a program P and a safety property t, the task of a model checker is to find a counter-example, that is, an execution of P that does not satisfy t, or to prove the absence of counter-examples on P. In the bounded symbolic model checking approach followed in the paper the model checker encodes P into a logical formula, conjoins it with the negation of t, and checks the satisfiability of the encoding using an SMT solver. If the encoding is unsatisfiable, the program is safe, and we say that t holds in P. Otherwise, the satisfying assignment the SMT solver found is used to build a counter-example.

Function Summaries. In HiFrog, function summaries are Craig interpolants [20]. The summaries are extracted from an unsatisfiable SMT formula of a successful verification, are over-approximations of the actual behavior of the functions, and are available for other HiFrog runs. We use the definition of function summaries [35] and SMT summaries [5] as in our previous works; examples of function summaries are available at `http://verify.inf.usi.ch/hifrog/tool-usage`.

HiFrog *and user-defined summaries.* The tool HiFrog [5] consists of two main components: an *SMT encoder* and an *interpolating SMT solver OpenSMT2* [29], and uses *function summaries* [34]. It is possible to provide to HiFrog a library of *user-defined summaries*, which are treated in the same way as function summaries by the SMT solver. We note that the whole set of summaries is uploaded to the SMT solver at once, which can lead to time-outs due to the formula being too large. In contrast, our approach by using lattices only uploads the subset of summaries that are necessary for solving the current instance of the library function. In the encodings of the experimental sections and examples we will use the quantifier-free SMT theories for equality logic with uninterpreted functions (EUF), linear real arithmetics (LRA), and fixed-width bit vectors. Note that fixed-width bit vectors are essentially propositional logic.

3 Construction of the Lattice of Facts

In this section we formally define the semilattice of facts for a given library function and describe an algorithm for constructing it; the inner function calls in the algorithm are explained at the end of Sec. 3.2. We note that while the size of the semilattice can be exponential in the number of the facts, the construction of the semilattice is done as a *preprocessing step* once, and the results are used for verification of all programs with this function.

3.1 Definitions

A fact for a library function g with its assumption is added to the set of facts F_g as $(assume(X) \wedge fact(g))$ expression, where X is a constraint on the domain of the input to g under which $fact(g)$ holds. For example, for the *modulo* function, we can have a fact $assume((a \geq 0) \wedge (n > 0)) \wedge a\%n \geq 0$. For every fact $(assume(X) \wedge fact(g))$, we add a fact $assume(\neg X) \wedge \mathbf{true}$ to F_g. As we discuss later, this is done in order to ensure that the lattice covers the whole domain of input variables for the function g.

Given a set of facts F_g for a library function g, the subset lattice $L(F_g)$ is constructed as defined in Sec. 2. The height of $L(F_g)$ is $|F_g| + 1$ by construction, and the width is bounded by the following lemma on the width of a subset lattice.

Lemma 1. *For a set S of size s, let $L(S)$ be the subset lattice of S. Then, the width of $L(S)$ is bounded by $\binom{s}{\lfloor \frac{s}{2} \rfloor}$.*

Proof. The bound follows from Sperner's theorem [9] that states that the width of the inclusion order on a power set is $\binom{s}{\lfloor \frac{s}{2} \rfloor}$.

Not all elements in $L(F_g)$ represent non-contradictory subsets of facts. For example, a fact $f_1 = assume((a > 0) \wedge (n > 0)) \wedge a\%n \geq 0$ and a fact $f_2 = assume(a = 0) \wedge a\%n = 0$ are incompatible, as the conjunction of their assumptions does not hold for any inputs. In addition, some elements are equivalent to other elements, as the facts are subsumed by other facts. We remove the contradictory elements from the lattice, and for a set of equivalent elements we leave only one element. We denote the resulting set by $L^{min}(F_g) \subseteq L(F_g)$, and the number of facts in an element E, as $\#E$ $(E \in L(F_g))$.

It is easy to see that $L^{min}(F_g)$ is a *meet semilattice*, since if two elements are in $L^{min}(F_g)$, they are non-contradictory, and hence their intersection (or an element equivalent to the intersection) is also in $L^{min}(F_g)$. In general, we do not expect the $\top$ element, representing the whole set F_g, to be in $L^{min}(F_g)$. Rather, there is a set of *maximal* elements of $L^{min}(F_g)$, each of which represents a maximal non-contradictory subset of facts of F_g; we denote the set of *maximal* elements of $L^{min}(F_g)$ as $maxL^{min}(F_g)$.

In the next subsection we describe the algorithm for constructing $L^{min}(F_g)$.

3.2 Algorithm

The construction of a meet semilattice of facts for a library function g given a set of conjunctions of facts and their constraints expressed as assume statements, is described in Alg. 1. The algorithm consists of five main components:

Construct a subset lattice from the input. For every statement and its assumption, we construct a fact f_g (line 1); given the set F_g of all facts, we construct a subset lattice $L(F_g)$ as defined in Sec. 3.1 (line 2).
Consistency check. For every element in the subset lattice we analyse the subset of facts corresponding to this element (lines 3-10); if the subset contains no contradictions (lines 6-7), we add the node to the meet semilattice (line 8).
Equivalence check. Remove equivalent elements from the meet semilattice (lines 11-20).
Cleanup. After the execution of the checks and removal of elements above, it is possible that in the resulting structure, an element has a single predecessor (lines 21-25). In this case, we unify the element with its predecessor (line 23). This process is repeated iteratively until all elements have more than one predecessor, except for the direct successors of the $\perp$ element.
Overlapping Assumes. Strengthen an assumption to avoid overlapping between elements (line 26).

The result of the algorithm is the meet semilattice $L^{min}(F_g)$, as defined in Sec. 3.1. Clearly, the exact $L^{min}(F_g)$ depends on the input set of statements, as well as on the theory. We note, however, that $L^{min}(F_g)$ can be used by the SMT solver with a different theory than the one in which it was constructed, as long as an encoding of the facts in SMT-LIB2 format with this logic exists. For example, the reduced meet semilattice in Fig. 2 can be used in EUF, even when its construction is done via propositional logic, since the encoding of f_1, f_2, and f_3 exists in EUF. Algorithm 1 invokes the following procedures:

- $\#E$: the number of facts in an element E (defined in Sec. 3.1);
- `buildSubsetLattice`: construct a *subset lattice* $L(F_g)$ given a finite set F_g of facts;
- `minimise`: given an element $E \in L(F_g)$, remove any fact $f_g \in E$ such as that $\exists E' \subset E.(\bigwedge_{f'_g \in E' - \{f_g\}} f'_g) \implies f_g$, starting from the smallest to the largest E' (i.e., remove a fact f_g if other facts in E imply f_g; that way, we minimise the size of the E);
- `checkSAT`(F): determine the satisfiablity of a formula F;
- `swap`(E_1, E_2): swap the current subset of facts in E_1 with E_2, while (roughly speaking) each element keeps its own edges;
- `immediateLower`(E): get all immediate predecessors of the element E;
- `immediateUpper`(E): get all immediate successors of the element E;
- `fixOverlapsAssume`$(L^{min}(F_g))$: for each meet element $E \in L^{min}(F_g)$, change the assumptions of E's immediate successors to fix any overlapping assumptions. $assume(X)$ of an immediate successor with a trivial fact is updated by intersecting with negations of all (original) assumes of the rest of the immediate upper elements of E, when removing any successor with (altered)

Algorithm 1: Construction of $L^{min}(F_g)$

Input : $facts = \{(X_1, Y_1), \ldots, (X_n, Y_n)\}$: set of pairs of assumptions and facts.
Output: $L^{min}(F_g)$

$F_g \leftarrow \bigcup_{(X,Y) \in facts} \{assume(X) \wedge Y, assume(\neg X) \wedge \textbf{true}\}$
$L(F_g) \leftarrow$ `buildSubsetLattice`(F_g)
foreach *element* $E \in L(F_g)$ **do**
    `minimise`(E) //remove facts that are generalised by other facts in E
    $Query \leftarrow \bigwedge_{f_g \in E} f_g$
    $\langle result, \_ \rangle \leftarrow$ `checkSAT`$(Query)$
    if *result is* **SAT then**
        Add E to $L^{min}(F_g)$
    end
end
foreach *two elements* $E_{lower}, E_{upper} \in L^{min}(F_g)$ *such that* E_{lower} *is lower than* E_{upper} **do**
    $Query \leftarrow \neg(\bigwedge_{f_g \in E_{lower}} f_g \iff \bigwedge_{f_g \in E_{upper}} f_g)$
    $\langle result, \_ \rangle \leftarrow$ `checkSAT`$(Query)$
    if *result is* **UNSAT then**
        if $\#E_{lower} < \#E_{upper}$ **then**
            `swap`(E_{upper}, E_{lower})
        end
        Remove E_{lower} from $L^{min}(F_g)$
    end
end
foreach *element* $E \in L^{min}(F_g)$ **do**
    if $(\#$*immediateUpper*(E) *is* $1) \wedge (\#$*immediateLower*$($*immediateUpper*$(E))$ *is* $1)$ **then**
        Remove E from $L^{min}(F_g)$
    end
end
$L^{min}(F_g) \leftarrow$ `fixOverlapsAssume`$(L^{min}(F_g))$
return $L^{min}(F_g)$

$assume(X)$ equals to $false$. $assume(X)$ of an immediate successor with facts in F_g is strengthen by intersecting with the negation of an $assume(X)$ of overlapping elements with facts in F_g.

In Fig. 2, for example, the $assume(X)$ statement of f_2 originally was $(n > 0)$ thus the *assume*s of f_1 and f_2 overlap over many values, e.g., when $a = n$; in the example in Fig. 2 we fix the *assume* of f_2 to avoid such overlapping.

4 Lattice-Based Bounded Model Checking

In this section we describe the Lattice-Based Counterexample-Guided Abstraction Refinement algorithm for verifying programs with respect to a safety property. We present a formal notation for the data structure we use in the refinement algorithm and show that the refinement algorithm is complete.

4.1 Definitions

For a program P and a safety property t such as that $P \cup \{t\}$ has functions which are missing the full definition in the current level of abstraction, we denote the set of all such functions in $P \cup \{t\}$ as G, thus $G = \{g_1, \ldots, g_m\}$. Each function $g \in G$ has a meet semilattice $L^{min}(F_g)$. The set of all meet semilattices of functions in G is $\mathcal{L}_G^{min} = \{L^{min}(F_{g_1}), \ldots, L^{min}(F_{g_m})\}$.

For each statement $s \in P \cup \{t\}$ with $g \in G$ function, we create an instance of $L^{min}(F_g)$. The set $\mathcal{L}_{G,K}^{min}$ is a set of all instances of all meet semilattices in $\mathcal{L}_G^{min}$. A meet semilattice instance $L_i^{min}(F_g) \in \mathcal{L}_{G,K}^{min}$ is the i-th instance of function g in $P \cup \{t\}$ where $1 \leq i \leq k_g$, and $k_g \in K$ is the number of instance of g in $P \cup \{t\}$. For simplicity of the description of the refinement, we assume each s has at most one function $g \in G$; if there is more than one g, one can write an equivalent code that guarantees this property. Note that Alg. 3, Alg. 4, Alg. 2 change instances of meet semilattices and not the meet semilattice itself; since each statement with a function g requires a different set of facts and thus must traverse the meet semilattice independently with its instance.

During Alg. 3, we mark elements $E \in L_i^{min}(F_g)$ as **Safe** and add any such E to the cut of $L_i^{min}(F_g)$. A cut of $L_i^{min}(F_g)$ is a set of all elements with an in-edge in the cut-set of the graph representation of $L_i^{min}(F_g)$. Possible cuts in the reduced meet semilattice in Fig. 2 can be: $\{\{f_1\}, \{f_2\}\}$ or $\{\{f_3\}\}$.

Definition 1. *Let $X_{L_i^{min}(F_g)} \subset L_i^{min}(F_g)$ be a subset of elements. We say $X_{L_i^{min}(F_g)}$ is a cut of $L_i^{min}(F_g)$ if all chains from $\emptyset$ to element(s) $E_{max} \in max L_i^{min}(F_g)$ contain at least one element in $X_{L_i^{min}(F_g)}$.*

We use the elements in the cut of $L_i^{min}(F_g)$ to show Alg. 2 is complete, when given the union of all assumptions of elements in $X_{L_i^{min}(F_g)}$ is capturing the whole domain of the inputs of g,

Lemma 2. *Given a cut $X_{L_i^{min}(F_g)}$ of function $g : \mathbb{D}_{in} \rightarrow \mathbb{D}_{out}$ the union of all assumptions (assume statements) of all facts in the cut is $\mathbb{D}_{in}$.*

Proof. We prove by induction that for a subset lattice $L(F_g)$: for any element $E \in L(F_g)$ its *assume* refers to the same domain as the union of *assume*s of all successors of E element.

(base) the union of *assume*s of all successors of $\emptyset$ element is $\mathbb{D}_{in}$: from line 1 in Alg. 1 we know that the union of *assume*s of all successors of $\emptyset$ element is $\mathbb{D}_{in}$ by construction of F_g, and $\emptyset$ element has no assumption and thus captures all the input domain.

(step) for each element $E \in L(F_g)$, the union of *assume*s of all successors of E is equivalent to the *assume* of E. Since $L(F_g)$ is a subset lattice, then all immediate upper elements of an element $E \in L(F_g)$ contain exactly one additional fact from F_g. From line 1 in Alg. 1, we know that any fact $(assume(X) \wedge Y)$ has the opposite fact ($\neg assume(X) \wedge \mathbf{true}$), thus union of any such pair of facts in F_g leaves the original *assume* of E the same; since each of the successor of E must contain either an original fact or its complementary fact, we get that the *assume* of the union of the successors of E stays the same as required.

Since all chains start from $\emptyset$ which refers to the whole domain $\mathbb{D}_{in}$, and since the *assume* of an element is a union of *assume*s of its immediate successors as proved by induction above, then if there is a cut where the union of all *assume*s of all facts in the cut is not $\mathbb{D}_{in}$ then there is a chain from $\emptyset$ to maximal element without an element in the cut, which contradict the definition of a cut. When extract $L^{min}(F_g)$, we only fix overlapping *assume*s thus the union of *assume*s stays the same in a cut and therefore refers to the whole domain as before. $\square$

Note that, the rest of the changes of elements in $L^{min}(F_g)$ do not affect the union of *assume*s; consistency check removes elements with no contribution to the input domain (as these equivalent to false), equivalence check affects only the number of possible cuts, and cleanup removes elements with the same *assume* with a weaker fact in compare to their single immediate successor.

4.2 Algorithm

Algorithm 2 takes the symbolically encoded program P with a safety property t and constructs an over-approximating formula $\hat{\varphi}$ of the problem in a given initial logic (line 1). Algorithm 2 refines $\hat{\varphi}$ by adding and removing facts from meet semilattices $L^{min}(F_g) \in \mathcal{L}_G^{min}$ according to the traversal on an instance of the meet semilattice per refined expression (main loop, lines 3-21); the algorithm terminates once it has proved the current $\hat{\varphi}$ is **Safe** (lines 8-10), after extracting a real counterexample (lines 14-16), or after using all facts in meet semilattices of $\mathcal{L}_G^{min}$ while still receiving spurious counterexamples (lines 17-19 or 23). The refinement in Alg. 2 is finite and returns **Unsafe** if t does not hold in P. Algorithm 2 returns **Safe** if and only if the facts in $\mathcal{L}_G^{min}$ can refine functions in $\hat{\varphi}$ and t holds in P.

A counterexample in the last known precision is returned when t does not hold in P and the facts in $\mathcal{L}_G^{min}$ can refine the over-approximate functions in $\hat{\varphi}$. Algorithm 2 checks if *CE* is a spurious counterexample similarly to the counterexample check in [28] and returns either true with a real counterexample when all queries are **SAT**, or false otherwise. The solver produces an interpretation for the variables or a partial interpretation of uninterpreted functions and uninterpreted predicates in the case of EUF, for statements $s \in P \cup \{t\}$ in the current precision. The counterexample validation determines whether the conjunction of s and *CE* with an interpretation or partial interpretation is **UNSAT** in a more precise theory; an **UNSAT** result in any of the queries indicates that the counterexample is indeed spurious. A more precise theory can be the theory of bit-vectors as in [28] or the theory the meet semilattice was built with; if no available description of the g with the current query exist in any preciser theory, we assume *CE* is spurious.

The data structures used in Alg. 2 are described in Sec. 4.1. Note that Alg. 2 allocates a new instance of a meet semilattice $L_i^{min}(F_g) \in \mathcal{L}_{G,K}^{min}$ for each i-th instance of function g in $P \cup \{t\}$, thus the main loop in lines 3-21 refers only to these instances of meet semilattices, where i, k_g, g, K, G are defined in Sec. 4.1.

Algorithm 2: Lattice-Based Counterexample-Guided Refinement

Input : $P = \{s_1 \coloneqq (x_1 = t_1), \ldots, s_n \coloneqq (x_n = t_n)\}$: a program, t: safety property, $\mathcal{L}_G^{min} = \{L^{min}(F_{g_1}), \ldots, L^{min}(F_{g_m})\}$: a set of meet semilattices.

Output: $\langle$**Safe**$, \perp\rangle$ or $\langle$**Unsafe**$, CE\rangle$ or $\langle$**Unsafe**$, \perp\rangle$

1 $\hat{\varphi} \leftarrow \bigwedge_{s \in P \cup \{t\}}$ `convert`(s)
2 $\mathcal{L}_{G,K}^{min} \leftarrow \bigcup_{s \in P \cup \{t\}, g \in G, i \in \{1,\ldots,k_g(\in K)\}} (L_i^{min}(F_g) \leftarrow$ `initialiseLI`$(s, L^{min}(F_g)))$
3 **while** $\exists L_i^{min}(F_g) \in \mathcal{L}_{G,K}^{min}$: *element*$(L_i^{min}(F_g))$ *has upper element* **do**
4     $\chi \leftarrow \bigwedge_{L_i^{min}(F_g)' \in \mathcal{L}_{G,K}^{min}}$ `currentFacts`$(L_i^{min}(F_g)')$
5     $Query \leftarrow \hat{\varphi} \wedge \chi$
6     $\langle result, CE\rangle \leftarrow$ `checkSAT`$(Query)$
7   **if** *result is* **UNSAT then**
8   **if** $(\forall L_i^{min}(F_g)'' \in \mathcal{L}_{G,K}^{min}$: *isSafe*$(L_i^{min}(F_g)'')) \vee (\chi$ *is* **true**$)$ **then**
9   **return** $\langle$**Safe**$, \perp\rangle$ //Safe - Quit
10   **end**
11         $\mathcal{L}_{G,K}^{min} \leftarrow$ `updateCutAndWalk`$(\mathcal{L}_{G,K}^{min})$ //element is safe, continue traversal
12   **end**
13   **else**
14   **if** *checkRealCE*$(Query, CE)$ **then**
15   **return** $\langle$**Unsafe**$, CE\rangle$ //Real Counterexample - Quit
16   **end**
17   **if** *!refine*$(Query, CE, P, t, \mathcal{L}_{G,K}^{min})$ **then**
18   **return** $\langle$**Unsafe**$, \perp\rangle$ //Cannot Refine - Quit
19   **end**
20   **end**
21 **end**
22 // End Of Main Loop
23 **return** $\langle$**Unsafe**$, \perp\rangle$ //Cannot refine - Quit

Sub-Algorithm 3 is a high-level description of `updateCutAndWalk` sub-procedure. For the current instance of a meet semilattice $L_i^{min}(F_g)$ where E is the current element, `updateCutAndWalk` marks E as safe, adds E to the cut of $L_i^{min}(F_g)$, and traverses on an instance of a meet semilattice via `walkRight` on either $L_i^{min}(F_g)$ (if not yet safe) or the last changed instance before $L_i^{min}(F_g)$; $L_i^{min}(F_g)$ is the last instance `walkRight` changed. Note that the sub-procedure `walkRight` changes the same instance of a meet semilattice until Alg. 2 is in either lines 9,15,18, or 22, or Alg. 3 is in lines 3-5.

A high-level description of the sub-procedure `refine` is given in Alg. 4, and describes the refinement of a single CE via instances of a meet semilattice. The

Algorithm 3: `updateCutAndWalk` - Mark element as safe and traverse the semilattice

Input : $\mathcal{L}^{min}_{G,K}$: a set of meet semilattice instances.
Output: $\mathcal{L}^{min}_{G,K}$ after traversal

$L^{min}_i(F_g) \leftarrow$ last changed meet semilattice instance in $\mathcal{L}^{min}_{G,K}$
Mark current element in $L^{min}_i(F_g)$ as **Safe**
if $\mathit{isSafe}(L^{min}_i(F_g))$ **then**
  $\forall L^{min}_i(F_g)' \in \mathcal{L}^{min}_{G,K}.\neg\texttt{isSafe}(L^{min}_i(F_g))' \implies \texttt{reset}(L^{min}_i(F_g)')$
  Set $L^{min}_i(F_g)$ to be an item from the set $\{L^{min}_i(F_g)'' | L^{min}_i(F_g)''' \in \mathcal{L}^{min}_{G,K} \wedge \neg\texttt{isSafe}(L^{min}_i(F_g))'''\}$
end
$\texttt{walkRight}(L^{min}_i(F_g))$
return $\mathcal{L}^{min}_{G,K}$ // Returns back to the main loop in Alg. 2 line 11

main loop (lines 1-13) searches $L^{min}_i(F_g)$ which refines CE, the inner loop (lines 3-9) adds facts from elements in $L^{min}_i(F_g)$ until CE is refined or a maximal element is reached; in the latter case we drop the changes in $L^{min}_i(F_g)$ (lines 10-12) and try a different $L^{min}_{i'}(F_{g'})$. The refinement successes if the query (line 5) detects $CE$is a spurious counterexample without using a more precise theory (lines 4-8) but using new added facts (line 10, previous loop). The refinement fails if for all $L^{min}_i(F_g) \in \mathcal{L}^{min}_{G,K}$, no element could refine the current CE (lines 17-19). The refinement order is determined by the way Alg. 4 goes over statements $s \in P \cup \{t\}$ (line 1), which is done according to sets of basic heuristics defined in [28].

We describe the rest of the function calls in general; let s be a statement $s \in P \cup \{t\}$, F be a logical formula, CE a counterexample, x a meet semilattice of a statements s with a function g, and x' an instance of a meet semilattice x. Algorithms 2, 3, and 4 invoke the following procedures:

- `convert`(s): create a symbolic formula in the initial logic;
- `checkSAT`(F): determine the satisfiablity of a formula F;
- `checkRealCE`(F, CE): is true if CE is a valid counterexample of formula F;
- `element`(x'): retrieve the current element in x' or $\top$ for x' with a full cut;
- `currentFacts`(x'): retrieves the formula of facts in x' which is either a union of all elements in the cut, an intersection of the facts in the current element, or *true* if the current element is the $\emptyset$;
- `walkRight`(x'): simulate a traversal of x' as described below;
- `walkUpper`(x'): simulate a traversal of x' from the current element to upper elements;
- `initialiseLI`(s, x): create an instance of a meet semilattice x' for s and `operation`(s), if a meet semilattice exists in L^{min}_G for `operation`(s); `operation`(s): retrieve the operation or function call name in s;
- `isSafe`(x'): indicate if x' refines g in s with **Safe** result as described above, an **Unsafe** result of the refinement is taken care in the loop itself and does not need a sub-procedure;

Algorithm 4: `refine` with a Single Counterexample

Input : *Query* and *CE* formulas, and $P = \{s_1 \coloneqq (x_1 = t_1), \ldots, s_n \coloneqq (x_n = t_n)\}$: a program, t: safety property, $\mathcal{L}^{min}_{G,K}$: a set of meet semilattice instances.
Output: *true* or *false*

for $s \in P \cup \{t\}$ *with* $L^{min}_i(F_g) \in \mathcal{L}^{min}_{G,K}$ **do**
  $n \leftarrow$ `element`$(L^{min}_i(F_g))$ //To reset later to original location
  while *element*$(L^{min}_i(F_g))$ *has upper element* **do**
    $\chi' \leftarrow$ `currentFacts`$(L^{min}_i(F_g))$
    $\langle result, \_\rangle \leftarrow$ `checkSAT`$(Query \wedge CE \wedge \chi')$
    if *result is* **UNSAT then**
      break // Refined the current *CE*
    end
    if *result is* **SAT then**
      `walkUpper`$(L^{min}_i(F_g))$
    end
  end
  if *element*$(L^{min}_i(F_g)) \in maxL^{min}_i(F_g) \wedge result$ *is* **SAT then**
    `reset`$(L^{min}_i(F_g), n)$
  end
end
if *all* $L^{min}_i(F_g) \in \mathcal{L}^{min}_{G,K}$ *reset location in line 11* **then**
  return *false* // Returns and terminates the main loop in Alg. 2 lines 17-18
end
return *true* // Returns back to the main loop in Alg. 2 line 17

– `reset`(x'): set the current element in the walk to be $\perp$ and initialise the inner state of the search on the meet semilattice instance.

Note that, the function `updateCutAndWalk` is Alg. 3, and the function `refine` is Alg. 4, both are been called in the main loop of Alg. 2, lines 11 and 17 respectively.

Traversal of a meet semilattice. For function g such as that g is over-approximated in the initial theory and g has a meet semilattice $L^{min}(F_g) \in \mathcal{L}^{min}_G$, the algorithm creates an instance of a meet semilattice $L^{min}_i(F_g)$ to simulate the traversal of the meet semilattice in a DFS style per instance of g. Several instances of a meet semilattice of g are required for example when g is part of a loop.

A traversal on an instance of a meet semilattice $L^{min}_i(F_g)$ starts with $\emptyset$ element, adding no facts to the query $\hat{\varphi}$. During execution, if $\hat{\varphi}$ is **SAT** in the current precision, then the next element on the traversal is one of the immediate successors of the current element, as long as no real counterexample is obtained, in which case the algorithm terminates and returns **Unsafe** with the counterexample. After reaching an element in $maxL^{min}_i(F_g)$ during the traversal indicates that the facts in the elements of $L^{min}_i(F_g)$ cannot refine the $i-th$ instance of

g with respect to the spurious counterexample, which can also terminate the refinement in Alg. 2 and returns **Unsafe**.

Once the query $\hat{\varphi}$ with facts of $E \in L_i^{min}(F_g)$ is **UNSAT**, the traversal skips the successors of E, marks E as safe, adds E to $X_{L_i^{min}(F_g)}$, and continues with one of the siblings of E according to the DFS order from left to right; if there are no remaining siblings of E, the traversal of $L_i^{min}(F_g)$ terminates, and outputs the cut $X_{L_i^{min}(F_g)}$; there is no use of a current element of the meet semilattice $L_i^{min}(F_g)$ once the traversal terminates and only the facts in its cut are used.

For a program with several instances of meet semilattices, once Alg. 2 finds a cut $X_{L_i^{min}(F_g)}$, the cut is added to χ' as a union of all elements in the cut with their facts. This allows using the facts in the cut for searching cuts on the rest of the instances of meet semilattices.

The following theorem shows that if Alg. 2 outputs a positive result (that is, the program is safe with respect to the given bound), then there are no counterexamples up to the given depth in the program.

Theorem 1. *Given a program P, a safety property t, a set of functions ($g \in$) G, and a set of instances of meet semilattices $\mathcal{L}_{G,K}^{min}$ for the functions in G, if there exists a cut $X_{L_i^{min}(F_g)}$ in the meet semilattice of facts $L_i^{min}(F_g)$ for each instance $i \in k_g$ of the function g such that the result of solving the program with each element in $X_{L_i^{min}(F_g)}$ is* **UNSAT***, then the program is safe with respect to the given bound and the property.*

Proof (Sketch). Alg. 2 returns **Safe** in line 8 when all $L_i^{min}(F_g)$ are safe with respect to their cuts $X_{L_i^{min}(F_g)}$. The last query (Alg. 2, line 6) just before satisfying the condition in line 8 is a conjunction of union of elements of cuts $X_{L_i^{min}(F_g)}$ of each of the instances of the meet semilattice. By Lemma 2, the union of *assume* statements of elements in the cut is the input domain $\mathbb{D}_{in}$ of g, for all instances $i \in k_g$ of all $g \in G$. Therefore, if no satisfying assignment has been found in the cut, there is no satisfying assignment in $\mathbb{D}_{in}$ of g, for all instances of g in the unwound program P. Therefore, the result is **UNSAT**, and the program is safe with respect to the given bound. □

The cut we use, is a disjunction (i.e., union of elements in a cut) of a conjunction of facts (i.e., intersection of all facts in an element in a cut); when using more than a single cut in *Query*, the expression is a conjunction of the expression of a cut above. The full proof of Theorem 1 is shown in App. A using a formal definition of the expression of a cut.

5 Implementation and Evaluation

This section describes the prototype implementation and the evaluation of the lattice-based counterexample-guided refinement framework.

The algorithm is implemented on the SMT-based function summarisation bounded model checker HiFrog [5] and uses the SMT solver OpenSMT [29]. The experiments run on a Ubuntu 16.04 Linux system with two Intel Xeon E5620

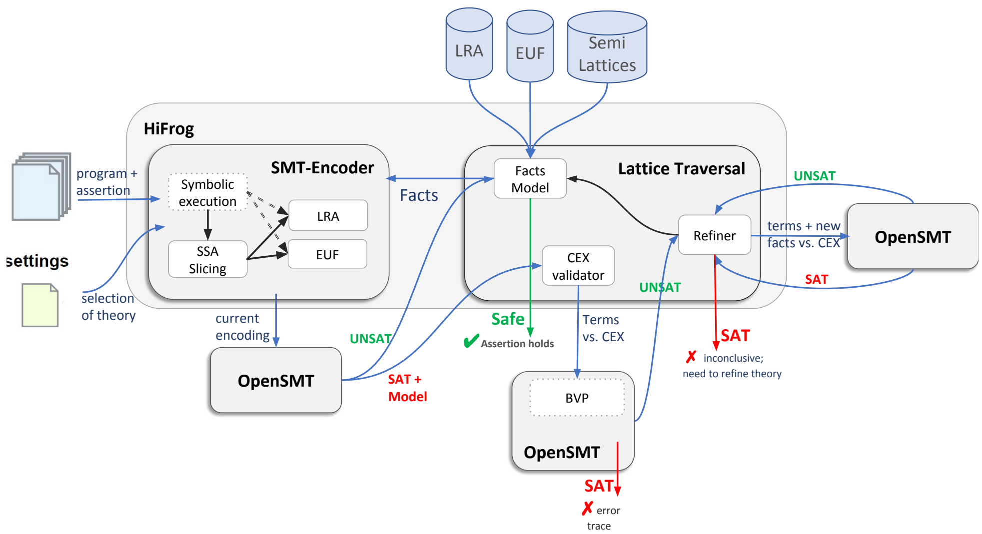

Fig. 3. The SMT-based model checking framework implementing a lattice-based counterexample-guided refinement approach used in the experiments.

CPUs clocked at 2.40GHz. The timeout for all experiments is at 500 s and the memory limit is 3 GB.

The scripts for the build of a meet semilattice, the meet semilattice for modulo operation, the complete experimental results, and the source code, are available at [1,2]. The script contains greedy optimisations of Alg. 1 to avoid, if possible, exponential number of SAT-solver calls; lines 3-10: starting the loop from the smallest subsets of facts, once a small subset of facts of an element is contradictory, all its upper elements are pruned; lines 11-19: considers only pairs of (roughly speaking) connected elements.

The overview of interaction between HiFrog, the refiner in HiFrog and the SMT solver OpenSMT is shown in Fig. 3. In the current prototype we add facts of the meet semilattice as SMT summaries, while checking before using a summary that its *assume* formula holds for better performance. The definition of the cut stays the same and contains only facts from F_{mod} where $g := mod$. The spurious counterexample check is done via the CEX validator using bit-vector logic (see [28]); any function that has no precise encoding is then added as a candidate to refine as HiFrog cannot validate a counterexample in the context of this function.

The lattice traversal component contains 3 sub-components: (1) *facts model* which contains the pure model (the meet semilattice) we load to HiFrog and instances of a meet semilattice per expression we refine, (2) the *CEX validator* that validates the counterexample and reports real counterexamples in case found, and (3) the *refiner* which does the refinement, adds facts to and removes facts from the encoding, interacts with the CEX validator and terminates the

Table 1. Verification results of lattice refinement against CBMC [15], theory refinement [28], and EUFand LRAwithout lattice refinement. #-number of instances, FP SAT-false positive SAT result, TO-time out of 500 s, MO-Out of Memory of 3GB.

Approach	#instances solved		#instances unsolved	
	SAT	UNSAT	FP SAT	TO,MO
LRA Lattice Ref.	**23**	32	9	**10,0**
EUF Lattice Ref.	**23**	8	33	**10,0**
Theory Ref.	22	18	20	11,3
CBMC 5.7	**23**	**34**	**1**	6,10
PURE LRA	**23**	7	34	**10,0**
PURE EUF	**23**	6	35	**10,0**

refinement for each of the three possible cases. The OpenSMT instances use either EUF or LRA for modeling and bit-vectors for CEX validation.

Extraction of facts. The preprocessing step of our framework is extracting a set of facts F_g for a function g. The facts can be imported from another program or a library. In the experimental results, we import facts from the Coq proof assistant [3], where $g := mod$ is modulo function. We use a subset of lemmas, theorems and definitions of modulo from [3] as is, as the data is simple to use, well known, and reliable. We translate the facts into the SMT-LIB2 format manually (see [1] for the results of translation).

Validation. The validation test is as follow; given a function g, a set of facts F_g, a statement s such that a fact $f_s \in F_g$ is sufficient to verify s, assure that $s \wedge f_s$ is **UNSAT** via a model checker. The validation tests can fail to assure $s \wedge f_s$ holds if too little facts were taken in the build stage (i.e., F_g is too small). A complementary validation test is the sanity check which verifies that the facts are not contradictory. We describe in details the validation tests for modulo operator in Appendix B; thus the function g is mod and the set of facts is $F_g := F_{mod}$.

Experimental Results We use a meet semilattice for refinement of modulo function with a set of 20 facts which are a small arbitrary subset of modulo operation properties; the width and the high of the modulo meet semilattice are 21 and 18 respectively; the raw data is taken from the Coq proof assistant [3] (see [1] for a meet semilattice sketch). The **UNSAT** proof of queries during the refinement are done using either OpenSMT[29] or Z3[21] using a none-incremental mode of solvers, due to known problems in the OpenSMTimplementation; we expect better experimental results in terms of time and memory consumption once improving the implementation.

We compared our lattice-based implementation on HiFrog (using both LRA and EUF encoding) with pure LRA and EUF encoding, theory-refinement mode of HiFrog, and cbmc version 5.7 the winner of the software model checking competition falsification track in 2017 (cbmc version 5.7 `--refine` option performs as the standard cbmc version, and thus is not included in Table 1). Our benchmarks consist of 74 C programs using the modulo operator at least few times; in 19 benchmarks the modulo operator is in a loop. The benchmarks set is a mix of 19 SVComp 2017 benchmarks [4] (8 **Unsafe** and 11 **Safe** benchmarks), our own 24 benchmarks including some hard arithmetic operations with modulo and multiplication, and 31 crafted benchmarks with modulo operator (20 **Unsafe** and 35 **Safe** benchmarks). Table 1 provides the summary of the experimental results.

Even with a prototype implementation of meet semilattices of facts, HiFrog fares quite well in comparison to established tools. In particular, it has better resource consumption than CBMC and theory-refinement mode of HiFrog, while also having much better results proving safety of programs than HiFrog without lattices. The lattice base refinement approach can still fail to prove safety when other operations are abstracted from the SMT encoding (e.g., SHL, SHR, pointer arithmetics) or, in LRA when the code contains non-linear expressions. Another reason is related to the modeling itself: a small sample of 20 can be insufficient to prove safety, as well the combination of several meet semilattices might require smarter heuristics. None of the approaches in the comparison reports **Unsafe** benchmarks as **Safe**. The full table of results and the set of benchmarks are available at [1].

Acknowledgments We thank Grigory Fedyukovich for helpful discussions.

References

1. `http://verify.inf.usi.ch/content/lattice-refinement`
2. `https://scm.ti-edu.ch/projects/hifrog/`
3. The coq proof assistant. `https://coq.inria.fr/`
4. Competition on software verification (SV-COMP). `https://sv-comp.sosy-lab.org/2017/` (2017)
5. Alt, L., Asadi, S., Chockler, H., Even Mendoza, K., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: HiFrog: SMT-based function summarization for software verification. In: Proc. TACAS '17. pp. 207–213. Springer (2017)
6. Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: A Proof-Sensitive Approach for Small Propositional Interpolants. In: VSTTE. LNCS, vol. 9593, pp. 1–18. Springer (2015)
7. Alt, L., Hyvärinen, A.E.J., Sharygina, N.: LRA interpolants from no man's land. In: Strichman, O., Tzoref-Brill, R. (eds.) Proc. HVC 2017. LNCS, vol. 10629, pp. 195–210. Springer (2017)
8. Alt, L., Hyvärinen, A.E.J., Asadi, S., Sharygina, N.: Duality-based interpolation for quantifier-free equalities and uninterpreted functions. In: Stewart, D., Weissenbacher, G. (eds.) Proc. FMCAD 2017. pp. 39–46. IEEE (2017)
9. Anderson, I.: Combinatorics of Finite Sets. Clarendon Press, Oxford (1987)
10. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Tools and Algorithms for the Analysis and Construction of Systems. LNCS, vol. 1579. Springer (1999)
11. Birkhoff, G.: Lattice Theory. AMS, 3rd edn. (1967)
12. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In: Legay, A., Margaria, T. (eds.) Proc. TACAS 2017. LNCS, vol. 10205, pp. 58–75 (2017)

13. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Satisfiability modulo transcendental functions via incremental linearization. In: de Moura, L. (ed.) Proc. CADE 2017. LNCS, vol. 10395, pp. 95–113. Springer (2017)
14. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV. LNCS, vol. 1855, pp. 154–169. Springer (2000)
15. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
16. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
17. Cousot, P.: Partial completeness of abstract fixpoint checking. In: Choueiry, B.Y., Walsh, T. (eds.) Abstraction, Reformulation, and Approximation. pp. 1–25. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
18. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977)
19. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 269–282. POPL '79, ACM, New York, NY, USA (1979)
20. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. In: J. of Symbolic Logic. pp. 269–285 (1957)
21. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
22. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
23. D'Silva, V., Purandare, M., Weissenbacher, G., Kroening, D.: Interpolant strength. In: Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 5944, pp. 129–145. Springer (2010)
24. Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., Barrett, C.W.: Smtcoq: A plug-in for integrating SMT solvers into coq. In: Majumdar, R., Kuncak, V. (eds.) Proc. CAV 2017. LNCS, vol. 10427, pp. 126–133. Springer (2017)
25. Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples, and refinements in abstract model-checking. In: Cousot, P. (ed.) Static Analysis. pp. 356–373. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
26. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM 47(2), 361–416 (Mar 2000)
27. Ho, Y.S., Chauhan, P., Roy, P., Mishchenko, A., Brayton, R.: Efficient uninterpreted function abstraction and refinement for word-level model checking. In: FMCAD. pp. 65–72. ACM (2016)
28. Hyvärinen, A.E.J., Asadi, S., Even-Mendoza, K., Fedyukovich, G., Chockler, H., Sharygina, N.: Theory refinement for program verification. In: Proc. SAT 2017. pp. 347–363. Springer International Publishing, Cham (2017)
29. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Proc. SAT 2016. LNCS, vol. 9710, pp. 547–553. Springer (2016)

30. Jancík, P., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Kofron, J., Sharygina, N.: PVAIR: Partial Variable Assignment InterpolatoR. In: FASE. LNCS, vol. 9633, pp. 419–434. Springer (2016)
31. Kutsuna, T., Ishii, Y., Yamamoto, A.: Abstraction and refinement of mathematical functions toward smt-based test-case generation. International Journal on Software Tools for Technology Transfer 18(1), 109–120 (2016)
32. Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: PeRIPLO: A framework for producing effective interpolants in SAT-based software verification. In: LPAR. LNCS, vol. 8312, pp. 683–693. Springer (2013)
33. Rummer, P., Subotic, P.: Exploring interpolants. In: Formal Methods in Computer-Aided Design (FMCAD), 2013. pp. 69–76. IEEE (2013)
34. Sery, O., Fedyukovich, G., Sharygina, N.: FunFrog: Bounded model checking with interpolation-based function summarization. In: ATVA. LNCS, vol. 7561, pp. 203–207. Springer (2012)
35. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based Function Summaries in Bounded Model Checking. In: HVC. LNCS, vol. 7261, pp. 160–175. Springer (2012)

A Completeness Proof of the Lattice-Based Counterexample-Guided Refinement Algorithm

In this section we present the full proof of Theorem 1, while following the same structure of the sketch of the proof in Sec. 4. We first define formally the expression of the facts in an element, in a cut with several elements and finally, in a conjunction of cuts; we use the later to build the last query, in which the solver returns **UNSAT** just before terminating the refinement in line 9 in Alg. 2. Then we prove by using Lemma 2 and the cuts of all the instances of meet semilattices that the algorithm is complete.

Proof. Let $\mathcal{X}_{G,K}$ be the set of (currently) all known cuts, thus

$$\forall L_i^{min}(F_g) \in \mathcal{L}_{G,K}^{min}.(X_{L_i^{min}(F_g)} \in \mathcal{X}_{G,K} \iff L_i^{min}(F_g) \texttt{ is safe})$$

following Alg. 3, line 2; thus $X_{L_i^{min}(F_g)}$ is the cut of $L_i^{min}(F_g)$ (Def. 1), and is unique (assumption on the implementation of lines 3-4, Alg. 3).

Alg. 2 returns **Safe** in line 8 when all $L_i^{min}(F_g)$ are safe with respect to their cuts $X_{L_i^{min}(F_g)}$, which is a union of elements of the cut of the i-th instance of the meet semilattice; this expressed over all elements in the cut as:

$$\Psi_{X_{L_i^{min}(F_g)}} = \bigvee_{E \in X_{L_i^{min}(F_g)}} \hat{f}_E \texttt{, for } X_{L_i^{min}(F_g)} \subseteq L_i^{min}(F_g),$$

where each element E contributes the following expression to the cut:

$$\hat{f}_E = \bigwedge_{f_g \in E \subseteq F_g} f_g \texttt{, for } E \in L_i^{min}(F_g),$$

and for $E = \bot$ then $\hat{f}_E$ is:

$$\hat{f}_\bot = true \texttt{, for } E = \bot \texttt{ and } E \in L_i^{min}(F_g).$$

Note that, if the cut of $L_i^{min}(F_g)$ is with a single element $\bot$, then

$$\Psi_{X_{L_i^{min}(F_g)}} = \hat{f}_\bot = true.$$

This can happen if there is no need to refine the i-th instance of function g to prove the problem encoded in $\hat{\varphi}$ is **Safe**.

The last query (Alg. 2, line 6) just before satisfying the condition in line 8 is a conjunction of union of elements of cuts $X_{L_i^{min}(F_g)}$ of each of the instances of the meet semilattice, thus χ in line 4 is:

$$\chi_{last} = \bigwedge_{\substack{\forall L_i^{min}(F_g) \in \mathcal{L}_{G,K}^{min} \\ X_{L_i^{min}(F_g)} \in \mathcal{X}_{G,K} \\ g \in G, i \in k_g \in K}} \Psi_{X_{L_i^{min}(F_g)}} = \bigwedge_{\substack{\forall L_i^{min}(F_g) \in \mathcal{L}_{G,K}^{min} \\ X_{L_i^{min}(F_g)} \in \mathcal{X}_{G,K} \\ g \in G, i \in k_g \in K}} \bigvee_{E \in X_{L_i^{min}(F_g)}} \hat{f}_E =$$

$$= \bigwedge_{\substack{\forall L_i^{min}(F_g)\in\mathcal{L}_{G,K}^{min} \\ X_{L_i^{min}(F_g)}\in\mathcal{X}_{G,K} \\ g\in G, i\in k_g\in K}} \bigvee_{E\in X_{L_i^{min}(F_g)}} \bigwedge_{f_g\in E} f_g = \bigwedge_{\substack{\forall L_i^{min}(F_g)\in\mathcal{L}_{G,K}^{min} \\ X_{L_i^{min}(F_g)}\in\mathcal{X}_{G,K} \\ g\in G, i\in k_g\in K}} \bigvee_{E\in X_{L_i^{min}(F_g)}} \bigwedge_{f_g\in E} (assume_{f_g} \wedge fact_{f_g}),$$

and $\hat{\varphi} \wedge \chi_{last}$ is **UNSAT**.

We split the **UNSAT** last query case into two; either we are on the first iteration, $\chi = \chi_{last} = \mathbf{true}$, $\hat{\varphi}$ is **UNSAT**, and there is no need for refinement or (else) there is a need for refinement, $\hat{\varphi}$ is **SAT**, $\chi = \chi_{last} \neq \mathbf{true}$ and $\hat{\varphi} \wedge \chi$ is **UNSAT**. The first case is trivial (true for model checking in general); we wish to prove that the algorithm is complete for the second case.

By Lemma 2, the union of *assume* statements of elements in the cut is the input domain $\mathbb{D}_{in}$ of g, for all instances $i \in k_g$ of all $g \in G$, and thus

$$assumes_{last} = \bigwedge_{\substack{\forall L_i^{min}(F_g)\in\mathcal{L}_{G,K}^{min} \\ X_{L_i^{min}(F_g)}\in\mathcal{X}_{G,K} \\ g\in G, i\in k_g\in K}} \bigvee_{E\in X_{L_i^{min}(F_g)}} \bigwedge_{f_g\in E} (assume_{f_g})$$

is **SAT** when assuming none of the domains are empty. Since also $\hat{\varphi}$ is **SAT** then by Lemma 2 also the conjunction $\hat{\varphi} \wedge assumes_{last}$ is **SAT**.

In addition, we know that $\chi_{last} \neq \mathbf{true}$ from Alg. 2, lines 17-19. Since the condition in line 17 was never satisfied (else the result of the whole check is **SAT**), then at least once the changes in line 7 in Alg. 4 were not reset, and hence there is one fact or more in the expression of χ_{last}.

Therefore, if no satisfying assignment has been found for $\hat{\varphi} \wedge \chi_{last}$, then there is no satisfying assignment in $\mathbb{D}_{in}$ of g, for all instances of g in the unwound program P for all $g \in G$. Consequently, the result is **UNSAT** due to the fact(s), and the program is safe with respect to the given bound. □

Table 2. Results of Validation Test of Facts.

	Total	Prop	Pure LRA	Pure EUF	Facts, LRA	Facts, EUF
Safe	26	15	0	0	24	8
Unsafe	27	27	27	27	27	27

B Validation Results of Modulo Facts

In this section we describe the validation results for the facts used in the construction of a meet semilattice of modulo operator. However, note that this is not part of the experiments of any of the algorithms presented in the paper. The validation part is merely for assuring the input is valid as possible; where the input for the construction of the meet semilattice, is a set of facts.

We build a benchmark set of 53 C-programs with a single assertion, with 26 safe and 27 unsafe instances. Majority of facts have at least one **UNSAT** and one **SAT** benchmarks. We use HiFrog with summaries such as that each fact is a summary of g, where g is modulo operator.

Table 2 describes the result of the validation experiments for the three modes of HiFrog [5]: propositional logic, LRA and EUF encodings. We use summaries for LRA and EUF; propositional logic supports modulo function without additional facts. The two last columns in Table 2 is the facts validation results for LRA and EUF. Propositional encoding verifies 15 out of 26 safe benchmarks. The remaining 11 benchmarks resulted in either a timeout, out of memory or a different result than expected. A different result is possible due to choice of implementation of modulo in C. The reason for a timeout or out of memory result, are expensive operations at the bit-level precision in the crafted benchmarks. Neither LRA nor EUF can express the modulo function, thus none of them shall terminate with a **Safe** result, as reported. However, the validation results of the facts with LRA shows that combining a fact with LRA coding allows verification of most cases, unless they contain nonlinear operators. The validation results of the facts with EUF reports only 8 instances as **Safe** (EUF's expected behavior on mathematical benchmarks). The second row of Table 2 investigates the unsafe benchmarks, corresponding to the sanity check. All facts are non-contradictory hence addition of facts does not change the result of verification of an **Unsafe** instance for LRA and EUF.