Theory Refinement for Program Verification

Antti E. J. Hyvärinen[1], Sepideh Asadi[1], Karine Even-Mendoza[2], Grigory Fedyukovich[3], Hana Chockler[2], and Natasha Sharygina[1]

[1] Università della Svizzera italiana, Switzerland
antti.hyvaerinen@usi.ch, sepideh.asadi@usi.ch, natasha.sharygina@usi.ch
[2] King's College London, UK
karine.even_mendoza@kcl.ac.uk, hana.chockler@kcl.ac.uk
[3] University of Washington, USA
grigory@cs.washington.edu

Abstract. Recent progress in automated formal verification is to a large degree due to the development of constraint languages that are sufficiently light-weight for reasoning but still expressive enough to prove properties of programs. Satisfiability modulo theories (SMT) solvers implement efficient decision procedures, but offer little direct support for adapting the constraint language to the task at hand. *Theory refinement* is a new approach that modularly adjusts the modeling precision based on the properties being verified through the use of combination of theories. We implement the approach using an augmented version of the theory of bit-vectors and uninterpreted functions capable of directly injecting non-clausal refinements to the inherent Boolean structure of SMT. In our comparison to a state-of-the-art model checker, our prototype implementation is in general competitive, being several orders of magnitudes faster on some instances that are challenging for flattening, while computing models that are significantly more succinct.

1 Introduction

The satisfiability modulo theories (SMT) [14] 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. Once a satisfying assignment is found for the propositional model, its consistency is queried as equalities from the theory solvers, which, in case of inconsistency, provide an explanation as a propositional clause. 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. Since in general more precise theories are both more expensive computationally and potentially distracting for the automatic reasoning, finding such a balance is a non-trivial task.

We introduce *theory refinement*, a counter-example-guided abstraction refinement (CEGAR) [11,12] approach for modeling software modularly using theories

that are partially ordered with respect to their precision. Our main contribution is the process of gradually encoding a program using the most precise theory only for a critical subset of all program statements, while keeping lower precision for the rest of the statements. The critical subset of theories is identified based on counter-examples, and theories of different precision are bound to each other through special identities. We study several automatic heuristics for guiding the encoding and provide also a manual encoding option. We apply theory refinement on verification of safety properties of software through bounded model checking. However, we believe that the technique is applicable in most verification techniques where higher level information is available on the problem structure. This includes model checking [5] and upgrade checking [15], k-induction [23], the IC3 algorithm [6], and generation of inductive invariants [16]. We show that the modular composition of the theories preferring lower precision can be used to both obtain speed-up in solving and identifying statements whose precise semantics do not affect the program safety, providing the model checker with cleaner proofs.

Many SMT solvers use over-approximation through theories as a means of speeding up solving. For instance [9,17,8] organizes the theory solvers into layers that solve problems represented in QF_BV. The query is first given to fast and less precise theory solvers, and only passed on to the exact solver if previous layers fail to show unsatisfiability. In contrast to low-level SMT solving, this work studies how to automatically identify statements whose exact semantics can be ignored in model-checking. This shift of view point has several advantages: (i) the approach can be used both to obtain speed-up in solving, and as a means for synthesis and finding fix-points for transition relations; (ii) the guidance from the source code allows the use of more powerful heuristics for choosing which statements should remain abstract; and (iii) the refinement takes place on the level of the program, not at the level of the theory query, an approach potentially more natural from the point of view of the semantics of the program.

We present theory refinement with two new theories called *uninterpreted functions for programs* (UFP) and *bit vectors for programs* (BVP) that are based on the theories of quantifier-free uninterpreted functions with equality (QF_UF), and bit vectors (QF_BV), respectively. The two theories were chosen since they represent two natural extremes in precision and are commonly used in the layered solver approach (see, e.g., [17]). In addition to the functionality of QF_UF, UFP provides interpretations for constants, conversion of abstract values to concrete values, and commutativity for uninterpreted functions when applicable. The key difference in BVP compared to QF_BV is that BVP is capable of directly injecting non-clausal refinements, modeling the program statements bit-precisely, to the inherent Boolean structure maintained in the SMT solver.

We implemented theory refinement on the SMT solver OpenSMT [19] and the bounded model checker HiFrog [3] supporting a subset of the C language. We report promising results both with respect to speed and the amount of refined program statements on both instances from a software verification competition and our own regression test suite. We demonstrate that the approach has a potential of several orders of magnitude of improvement over the approach based

solely on flattened bit-vectors, as implemented in the state-of-the-art tool CBMC and in our own tool. The implementation and the benchmarks are available at [1].

Related Work. Solving bit-vector problems with layers of theory solvers is introduced in [9] and further developed in [17]. While we work directly on software verification instead of bit-vectors, our approach is related, as we also use hierarchy of solvers combined with rewriting techniques. However, we work explicitly on the modeling language by automatically adjusting the precision to be different in different parts of the problem, and adding additional constraints that seams these parts together. In [8] a CEGAR based approach is used for solving problems involving arrays by transforming an abstract representation into clauses. We differ from this approach in that we integrate the system on the theory solver level, employing in the experiments the congruence closure algorithm together with a propositional solver. To the best of our knowledge, no existing approach uses this level of granularity in the modeling. Furthermore, we use counter-examples that are checked against the bit-precise implementation, and this way can avoid refinement of program parts that would need to be refined in approaches based on layered theory solvers.

Exploiting simultaneously several theories for one verification goal is not new. For example, [16] presents a system for synthesizing safe bit-precise inductive invariants for software. Compared to our work, the refinement direction is inverted: the software is first flattened, and in case of a time-out, converted to a domain-specific theory. Furthermore, we integrate seamlessly the theories UFP and BVP into an SMT solver whereas [16] considers real arithmetics.

Uninterpreted functions have been used together with the bit-precise encoding for verifying the equivalence of Verilog designs in [18,7]. The approach uses machine learning to identify sub-components that can likely be abstracted. In contrast, our emphasis is on software verification and integration to the SMT solver. A related approach [22] constructs test cases for scientific software by computing difference constraints from non-linear mathematical functions. This approach can be viewed as a special case of the framework we present in this paper; the formulas we derive can also be used for generating test cases, although this is not the focus of this paper. Similarly, [10] combines linear real arithmetic and equality of uninterpreted functions (QF_UF) for the SMT encoding of the program. The algorithm initially uses QF_UF to abstract non-linear operators, and then uses the monotonicity and the multiplication checks to identify spurious counterexample thus avoiding simulation and code execution. Both checks might result in a refinement formula, which is added then to the current SMT encoding. Unlike ours, their approach cannot be applied as such for bit-precise reasoning. In [3] we report early, very positive results on using the combination of EUF, LRA, and propositional flattening for encoding model checking problems. The current work which explores the possibilities in much more depth and rigor is motivated by this early result.

Another program-based refinement approach was proposed in [20], where compositional program is approximated with a program-specific theory of tran-

sition systems. Our approach is orthogonal to this, as we are able to handle programs in a more general way through the eventual flattening, while the theory of transition systems could likely be integrated as an additional theory.

2 Preliminaries

Let P be a loop-free program represented as a transition system, and t a *safety property*, 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 prove the absence of counter-examples on P. In the bounded, symbolic model checking approach followed in this 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.

A *sort* is a set of constants. For example the Boolean sort $\mathbb{B} = \{\top, \bot\}$ consists of the Boolean constants, true and false. Given a set of sorts $\{T_0, \ldots, T_n\}$, a *function* $op : T_1 \times \ldots \times T_n \to T_0$ maps a (possibly empty) sequence of constants $v_1, \ldots, v_n$ such that $v_i \in T_i$ to a *return value* $v_0 \in T_0$. Functions mapping empty sequences are *variables*, and a *term* is either a constant, a variable, or an application of a function $op(t_1, \ldots, t_n)$ where t_i are, recursively, terms with a return value in the sort T_i. In most cases in this paper we use the usual infix notation together with parentheses to express the well-known arithmetic and logical functions.

3 Combination of Theories in Theory Refinement

This section fixes a notation for describing instances of the safety problem using SMT, and provides two communicating theories for solving the safety problem. The goal of the presentation is to clarify how the modeling works in the SMT framework, placing particular emphasis to the use of symbols and their semantic.

In modeling programs we consider sets of quantifier-free symbolic statements of the form $x = t$, where x is a variable, and t is a term. This form essentially corresponds to the Single static assignment (SSA) form [13] for loop-free programs. The symbolic statements are defined over a sort of bounded integers Sz and a Boolean sort $Sb = \{\top_l, \bot_l\}$; we distinguish between these sorts and, for instance, the sorts of integers $\mathbb{Z}$ and Booleans $\mathbb{B}$ to clarify the difference between this *symbolic encoding* (hence the S) and the representation used by an SMT solver. Table 1 lists the non-variable functions we consider in our encoding. Note that unlike some programming languages, including C and C++, we do not allow the encodings to interpret terms from Sz as terms from Sb or vice versa. We distinguish between the functions defined over the sort Sb and those defined over Sz, calling the former logical functions and the latter non-logical functions.

Table 1. The functions used in the encoding we consider. Note that unsigned and signed sum coincide.

Functions		**Descriptions**
	Logical functions	
`&&`, `||`	$Sb \times Sb \to Sb$	Logical and, or
`!`	$Sb \to Sb$	Logical not
	Non-logical functions	
`+`, $*_u$, $*_s$, $/_u$, $/_s$	$Sz \times Sz \to Sz$	Sum, unsigned and signed product and division
$\%_u$, $\%_s$	$Sz \times Sz \to Sz$	Unsigned and signed remainder
$\ll, \gg_a, \gg_l$	$Sz \times Sz \to Sz$	left shift, arithmetic and logical right shift
`&`, `|`, `^`	$Sz \times Sz \to Sz$	Bitwise and, or, exclusive or
$\sim$:	$Sz \to Sz$	bitwise complement
$\leq_s$, $\leq_u$, $<_s$, $<_u$, $\geq_s$, $\geq_u$, $>_s$, $>_u$	$Sz \times Sz \to Sb$	Signed and unsigned less than or equal to and greater than or equal to

$$
\begin{array}{ll}
 & \left(c^b =_{BVz} \left((a^b \,\%_u\, 2^b) + (b^b \,\%_u\, 2^b)\right) \,\%_u\, 2^b\right)_1 \wedge \\
c = \left((a \,\%_u\, 2) + (b \,\%_u\, 2)\right) \,\%_u\, 2 & \left((c')^b =_{BVz} (a^b + b^b) \,\%_u\, 2^b\right)_1 \wedge \\
c' = (a + b) \,\%_u\, 2 & \left(d^u = f^u *_u e^u *_u c^u\right) \wedge \\
d = f *_u e *_u c & \left((d')^u = e^u *_u f^u *_u (c')^u\right) \wedge \\
d' = e *_u f *_u c' & \left(c^u = (c')^u\right) \leftrightarrow \left(\left(c_1^b \leftrightarrow (c')_1^b\right) \wedge \ldots \wedge \left(c_{bw}^b \leftrightarrow (c')_{bw}^b\right)\right)
\end{array}
$$

Fig. 1. (*Left*) a sequence of statements and (*right*) the corresponding encoding in combined UFP and BVP (to be described in Sect. 3.3). On the left all the variables are of sort Sz, and e and f are unbound.

The control-flow structures, such as `if-then-else`s, are encoded using the functions `!`, `||`, and `&&`. For the purpose of this presentation we assume that the encodings do not contain arrays and pointers.[4] Fig. 1 (*left*) shows an example sequence of statements that we will use as a running example in the discussion of this section.

[4] We do support these in our implementation, but their results are treated nondeterministically, that is, as unbound variables from Sz.

3.1 Bit Vectors for Programs

Our theory of bit vectors for programs (BVP) has a single sort BVz^{bw} containing the integers representable in $bw \in \mathbb{N}$ bits. When the bit-width of the sort is clear from the context we simply write BVz for the sort. Each BVP term t of sort BVz^{bw} is associated with the bits $t_1, \ldots, t_{bw}$ which are variables from the sort $\mathbb{B}$. The bits t_1 and t_{bw} are called, respectively, the *least significant bit* and the *most significant bit* of t.

The BVP theory has two special constants 1^b and 0^b. For the constant 0^b, $0^b_i = \bot$, $1 \leq i \leq bw$. For the constant 1^b, $1^b_1 = \top$ and $1^b_i = \bot$ for $2 \leq i \leq bw$. The equality of BVP is $=_{BVz}$: $BVz \times BVz \to BVz$. The interpretation of the equality is that if $x =_{BVz} y$ holds, then the value of the equality term is 1^b and otherwise 0^b. Finally, BVP has the functions defined in Table 1 with all sorts replaced by the sort BVz. For a term t, the Boolean functions determining the bits t_i are computed through propositional flattening (see, e.g., [21]).

We encode a sequence of statements $P = \{x_1 = t_1, \ldots, x_n = t_n\}$ in BVP as follows. Each statement $x_i = t_i$ is converted to $|x_i|^b =_{BVz} |t_i|^b$, where the operator $|\cdot|^b$ is defined for a symbolic term t recursively:

$$|t|^b \stackrel{\text{def}}{=} \begin{cases} x^b & \text{if } t \doteq x \text{ is a variable or a constant} \\ |x|^b \bowtie |y|^b & \text{if } t \doteq x \bowtie y \text{ where } \bowtie \text{ is a binary function,} \\ \circ|x|^b & \text{if } t \doteq \circ x \text{ where } \circ \text{ is a unary function} \end{cases} \tag{1}$$

where $a \doteq b$ denotes that the term a matches the form of b. Conjunction of the least significant bits of encoded statements in P defines its BVP-encoding $[P]^b$:

$$[P]^b \stackrel{\text{def}}{=} (|x_1|^b =_{BVz} |t_1|^b)_1 \wedge \ldots \wedge (|x_n|^b =_{BVz} |t_n|^b)_1 \tag{2}$$

We say that a safety property t holds in program P if and only if $[P]^b \wedge \neg[t]^b_1$ is unsatisfiable. Based on the definition we can see that the symbolic encoding in Fig. 1 satisfies the safety property ($d = d'$) due to properties of modular arithmetics. The BVP encoding is often inefficient due to the quadratic growth of the formula with respect to bw. However, in many cases, the bit-precise encoding of statements (e.g., $*_u$ in Fig. 1) are irrelevant to the safety property, and can therefore be over-approximated. This motivates the use of less precise but more efficiently solvable encodings such as those based on uninterpreted functions.

3.2 Uninterpreted Functions for Programs

The logic UFP (Uninterpreted Functions for Programs) is the standard logic of quantifier-free uninterpreted functions having the Boolean sort $\mathbb{B}$, the standard Boolean functions $op : \mathbb{B} \times \ldots \times \mathbb{B} \to \mathbb{B}$ where op is an operator such as $\vee, \wedge$, and $\neg$, and an unbounded number of variables. In addition the logic is augmented with

- a sort *UFPn* of real or integer numbers;
- the functions listed in Table 1 treated as uninterpreted functions with the sorts *UFPn* and $\mathbb{B}$ instead of *Sz* and *Sb* respectively;

– commutativity of the functions `+`, $*_u$, $*_s$, `&`, and `|`; and
– the concept of constants beyond the Boolean $\top$ and $\bot$.

As usual, UFP also contains the equality function $=_S\colon T \times T \to \mathbb{B}$ for all sorts T. As in the symbolic encoding, also in UFP we differentiate between two types of functions: those with a return sort $\mathbb{B}$, and those with a return sort *UFPn*.

Given a sequence of statements $P = \{x_1 = t_1, \ldots, x_n = t_n\}$, we denote its encoding in UFP by $[P]^u \stackrel{\text{def}}{=} ([x_1]^u =_{T_1} [t_1]^u) \wedge \ldots \wedge ([x_n]^u =_{T_n} [t_n]^u)$, where T_i is either *UFPn* or $\mathbb{B}$ depending on the related sort. The encoding operator $[\cdot]^u$ is defined as follows for a term t:

$$[t]^u \stackrel{\text{def}}{=} \begin{cases} x^u & \text{if } t \doteq x \text{ is a variable or a constant} \\ [x]^u \wedge [y]^u & \text{if } t \doteq x \,\texttt{\&\&}\, y \\ [x]^u \vee [y]^u & \text{if } t \doteq x \,\texttt{||}\, y \\ \neg[x]^u & \text{if } t \doteq \,\texttt{!}\, x \\ [x]^u \bowtie [y]^u & \text{if } t \doteq x \bowtie y \text{ where } \bowtie \text{ is a non-logical function.} \end{cases} \tag{3}$$

We distinguish between the notions of program safety in UFP and in BVP. In particular, we say that a safety property t holds in program P in UFP if and only if $[P]^u \wedge \neg[t]^u$ is unsatisfiable.

The program in Fig. 1 is safe with respect to the safety property $\texttt{!}\,(c = c')\,\texttt{||}\,(d = d')$ in UFP and therefore also in BVP. However, it is not safe in UFP with respect to the safety property $d = d'$ that is safe in BVP. For checking safety of programs in UFP we use a theory solver implementing a congruence closure algorithm [14] that is modified to support constants and commutativity. The modifications are described in more detail in Sec. 5.1.

In our recent experiments [3] we showed that safety of many programs can be established by interpreting the arithmetic functions as uninterpreted functions. In the next subsection we describe how the UFP logic and the BVP logic can be combined.

3.3 Combination of UFP and BVP

We present the theory refinement approach using a seamless integration of the UFP and BVP encoding, and therefore require a form of theory combination. However, unlike in conventional theory combination on bit vectors (see, e.g., [17]), we do not need to consider bit-vectors as theories, but instead they are embedded directly to the Boolean structure of the SMT solver. The two theories UFP and BVP are combined using a *binding formula* defined as follows.

Definition 1. *Given a symbolic statement t, let $[t]^u$ and $[t]^b$ be its* UFP *and* BVP*-encodings respectively. If both $[t]^u$ and $[t]^b$ appear together in a formula, we say that t is* bound*. Let B be the set of all bound statements. The* binding formula *for B (denoted F_B) is defined as*

$$F_B \stackrel{\text{def}}{=} \bigwedge_{t,t' \in B} ([t]^u = [t']^u) \leftrightarrow \left(([t]^b_1 \leftrightarrow [t']^b_1) \wedge \ldots \wedge ([t]^b_{bw} \leftrightarrow [t']^b_{bw})\right) \tag{4}$$

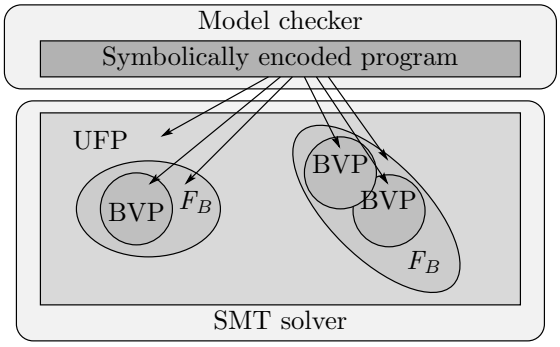

Fig. 2. A symbolic encoding of a program and the corresponding SMT formula. In the schematic example most of the program is encoded using UFP, while certain critical parts are encoded in BVP and made to communicate with the UFP encoding using the binding formula F_B.

Intuitively, the combination of the theories UFP and BVP with F_B allow us to express an over-approximation of the symbolic encoding of a program. This is stated more formally in the following theorem.

Theorem 1. *Let P be a program. Then $[P]^b \wedge F_B \models [P]^u$.*

Proof. (*sketch*) By simulation of executions in BVP: if there exist values $v_1^b, \ldots, v_n^b$ for the variables $x_1^b, \ldots, x_n^b$ in a term $[a = t]^b$ then the same values $v_1^u, \ldots, v_n^u$ satisfy the corresponding equality $[a]^u = [t]^u$. □

Fig. 2 shows the combined UFP and BVP encoding schematically. The symbolic encoding of a program is partitioned by the model checker into three parts: the UFP encoding, the BVP encoding, and the binding formula F_B. The conjunction of these is solved by the SMT solver. Fig. 1 (*right*) describes a combination encoding of UFP and BVP together with the necessary binding formula for the running example.

4 Counterexample-Guided Theory Refinement

This section provides an algorithm for verifying safety of programs by gradually refining the *precision* ρ of the symbolic encoding from UFP to BVP in parts where satisfying truth assignments show that it is necessary for soundness. Algorithm 1 describes the high-level idea. The algorithm takes as input a symbolically encoded problem P and a safety property t, and returns either **Safe**, if t holds in P, or **Unsafe** with a bit-precise counter-example if t does not hold in P. During the execution the algorithm picks statements $s \in P \cup \{t\}$ and refines their approximations in ρ until $\rho[s]$ is equivalent to $[s]^b$. Based on ρ, the algorithm constructs the binding formula F_B sufficient to connect the UFP and BVP terms.

Algorithm 1: The Counterexample-Guided Theory Refinement Algorithm

input : $P = \{(x_1 = t_1), \ldots, (x_n = t_n)\}$: a program, and t: a safety property
output: $\langle$**Safe**, $\bot\rangle$ or $\langle$**Unsafe**, $CE^b\rangle$

```
1  For all 1 ≤ i ≤ n initialize ρ[x_i = t_i] ← [x_i = t_i]^u
2  ρ[t] ← [t]^u
3  F_B ← ⊤
4  while true do
5  |   Query ← ρ[x_1 = t_1] ∧ ... ∧ ρ[x_n = t_n] ∧ ¬ρ[t] ∧ F_B
6  |   ⟨result, CE⟩ ← checkSAT(Query)
7  |   if result is UnSAT then
8  |   |   return ⟨Safe, ⊥⟩
9  |   end
10 |   CE^b ← getValues(CE)
11 |   foreach s ∈ P ∪ {t} s.t. ρ[s] ⊭ [s]^b do
12 |   |   ⟨result, _⟩ ← checkSAT([s]^b ∧ CE^b)
13 |   |   if result is UnSAT then
14 |   |   |   ρ[s] ← refine^s(ρ[s])
15 |   |   |   F_B ← computeBinding(ρ)
16 |   |   |   break
17 |   |   end
18 |   end
19 |   if No s was refined at line 14 then
20 |   |   return ⟨Unsafe, CE^b⟩
21 |   end
22 end
```

The safety of the program is tested at lines 5–9 using the current precision ρ and the binding formula. If the check succeeds, the algorithm terminates at line 9. Otherwise, a satisfying truth assignment is extracted at line 10 and then used to refine ρ at lines 11–18.

The need for refinement is checked for every statement s with a precision $\rho[s]$ not equivalent to $[s]^b$. If the truth assignment CE^b is inconsistent with $[s]^b$ then $\rho[s]$ is refined to block the truth assignment. If at least one such replacement happens in the current iteration, the execution proceeds to line 5. In practice it is a good idea to refine several statements based on a single counter-example, as discussed in Sec. 6. If no refinement is done, the truth assignment corresponds to a counter-example and the algorithm terminates at line 20.

The algorithm uses four sub-procedures `checkSAT`, `getValues`, `refine`^{s}, and `computeBinding`. `checkSAT`(F) determines the satisfiability of a formula F, `getValues`(CE) computes a BVP encoding of CE through substituting the abstract values from UFP with concrete BVP values. `refine`$^s(F)$ refines the statement s with respect to the previous precision F, and `computeBinding`(ρ) computes the binding formula using Def. 1. Below we give a definition for the `refine` procedure, while the other procedures will be discussed in more detail in Sec. 5.3.

Definition 2. *The procedure* $\texttt{refine}^s(F)$ *returns an iterative refinement of the statement* s *of the symbolic encoding with respect to* F*, such that (i)* $\texttt{refine}^s(F) \models F$*, and (ii)* $\texttt{refine}^s$ *has a fix-point that is equivalent to* $[s]^b$ *and reachable in a finite number of applications of* $\texttt{refine}^s$*.*

While in the implementation discussed in Sect. 5 we use $\texttt{refine}^s(F) = [s]^b \wedge [s]^u$, we want to point out the possibility of using interpolation-based methods (see, e.g., [4]) for the refinement.

Theorem 2. *Alg. 1 terminates in a finite number of steps.*

Proof. Assume that Alg. 1 does not terminate. Then there is a term in $P \cup \{t\}$ that can be refined an unbounded number of times before the fix-point equivalent to $[s]^b$ is reached, which contradicts Def. 2. □

Theorem 3. *Alg. 1 returns* **Unsafe** *if and only if the symbolic encoding* P *has an execution violating the safety property* t*.*

Proof. The algorithm maintains the invariants

$$\begin{array}{l} \text{Inv1 } [x_1 = t_1]^b \wedge \ldots \wedge [x_n = t_n]^b \models \rho[x_1 = t_1] \wedge \ldots \wedge \rho[x_n = t_n] \\ \text{Inv2 } [t]^b \models \rho[t] \end{array} \tag{5}$$

at line 14 by Def. 2 and Th. 1. Assume that the algorithm returns **Unsafe** but there is no execution violating the safety property t. Then there is a truth assignment σ such that $\rho[x_1 = t_1] \wedge \ldots \wedge \rho[x_n = t_n] \wedge F_B$ is true and $\rho[t]$ is false. The truth assignment σ must also satisfy $[x_1 = t_1]^b \wedge \ldots \wedge [x_n = t_n]^b$. By Inv2, if $\rho[t]$ is false also $[t]^b$ is false, hence contradicting the unsafety of (P, t). Now assume the algorithm returns **Safe** but there is an execution of P violating t. Then there is a truth assignment satisfying $[P]^b \wedge \neg[t]^b$. Since by Th. 1 both $[P]^b \wedge F_B \models \rho[x_1 = t_1] \wedge \ldots \wedge \rho[x_n = t_n]$ and $\neg[t]^b \wedge F_B \models \neg\rho[t]$, also the query on line 5 is satisfiable, contradicting the assumption. □

5 Implementation of Theory Refinement Algorithm

This section describes the prototype implementation of the theory refinement algorithm. The algorithm is implemented on the SMT solver OpenSMT [19] and the bounded model checker HiFrog [3]. The overview of implementation including the three main components and interactions between them is depicted in Fig. 3.

5.1 The Solver for UFP

The UFP theory solver is based on the co-operation between a congruence closure algorithm, which maintains sets of equivalence classes and inequalities between the classes, and a SAT solver, which enforces a propositional structure describing the relations between the equalities. We refer the reader to [14] for the full description of the *egraph* algorithm that the UFP solver bases on.

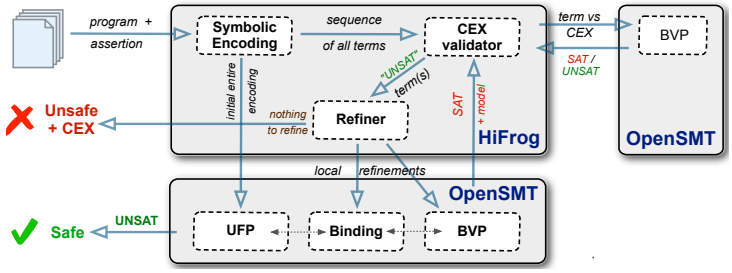

Fig. 3. The SMT-based model checking framework implementing a theory refinement approach used in the experiments.

Constants. The original egraph algorithm does not support constants other than the Boolean $\top$ and $\bot$, but constants play often an important role in our benchmarks. The egraph algorithm can represent an inequality between two terms t_1, t_2 by asserting explicitly the inequality $t_1 \neq t_2$ over these terms. This representation grows quadratically in the number of constants and therefore is not scalable. We adopt a different strategy for representing the inequalities between constants. An equivalence class in the egraph algorithm is represented by a linked list binding together the terms in the same class. Each class is represented by a canonical term from the linked list. In the original algorithm of [14], when two equivalence classes a and b are joined, the canonical term of the new class $a \cup b$ is the representative of whichever class a or b contains more terms. This is done to allow efficient joining and splitting in the backtracking search driven by the SMT solver. In our implementation the representative of a class a is always a constant if a contains a constant. The implicit inequality between constants is then implemented by a check that the respective equivalence classes are not both represented by a constant term. This approach fits naturally into the egraph algorithm and explanation generation. In the experiments we observed no noticeable slowdown compared to the original approach.

Values. Alg. 1 requires concrete values from the UFP theory to construct a counter-example candidate. In general the values for UFP are obtained by assigning a running number for each equivalence class that the egraph algorithm maintains. However, there are two special cases for the values. First, if the equivalence class contains a constant, the value is that of the constant. Second, a preprocessing step in the SMT solver removes terms that only appear on clauses that are true by construction. Since these terms can have any value, we indicate this with a special flag.

Commutativity. The commutativity of the functions $Co = \{ \texttt{+}, \texttt{*}_u, \texttt{*}_s, \texttt{\&}, \texttt{|} \}$ is implemented by conjoining the set $\{\circ(a,b) \leftrightarrow \circ(b,a) \mid \circ \in Co, \circ(a,b) \text{ in } P\}$ to the instance $[P]^u$ being solved. A similar approach is followed, for instance, in [10].

5.2 The Solver for BVP

The BVP theory is solved through propositional flattening [21]. The solver supports the operations listed in Table 1, and allows the use of arbitrary bit-widths.[5] Based on an extensive testing the implementation is robust, but still prototypical in the sense that we implement no sophisticated pre-processing techniques that are available in many other bit-vector solvers (see, e.g., [9]).

Unlike many other SMT solvers (see, e.g., [17]), we do not implement the bit-vector solver as a separate SAT solver working on the flattening and driven by the main SAT solver. Instead, we flatten the problem directly to the main SAT solver. This has several advantages: we avoid the overhead of duplicate solver instantiation, and we enable the solver to potentially learn much more intricate relationships between the flattened formula and the formula in UFP. However, an in-depth analysis of the implications of this design is beyond the scope of this paper.

5.3 Theory Refinement in Model Checking

We integrated Alg. 1 into the bounded model checker HiFrog for C programs. HiFrog obtains first the symbolic encoding of the program P and a safety property t through a sequence of pre-processing steps, builds then the UFP formula, and finally gradually transforms parts of the UFP formula into BVP based on truth assignments until the safety is determined. We follow the approach where safety properties are expressed as assertions in the C code. The architecture is depicted in Fig. 3. HiFrog maintains two SMT solvers during the execution and which are represented by the `checkSAT` calls in Alg. 1: the *main solver* for checking the satisfiability query constructed at line 5 (shown on the bottom of Fig. 3) and the *refinement solver* for checking the spuriousness of each counter-example at line 10 (shown on the right of Fig. 3). This choice was taken so that the expensive calls on the main solver would not be slowed down by unnecessary clauses at the refinement solver.

The counter-examples are flattened to propositional logic through the call to `getValues` by mapping the values in UFP to a unique bit-vector constant of the given bit width bw. At this stage of the development we ignore the case where the UFP solver gives more equivalence classes than what is representable in bw bits, since this limitation did not affect our results.

The binding formula (see Def. 1) is updated whenever a statement $x = t$ is refined. This is done by first constructing the BVP formulas $[x]^b$ and $[t]^b$, and then adding the missing equalities to F_B with the call to `computeBinding`.

6 Experimental Results

We evaluated the theory-refinement mode of HiFrog on C programs mostly coming from the software model checking competition (SV-COMP). The benchmarks were split into the *safe* (128 instances) and *unsafe* (30 instances) sets,

[5] The shift operations $\ll$, $\gg_a$, $\gg_l$ assume a bit-width that is a power of two.

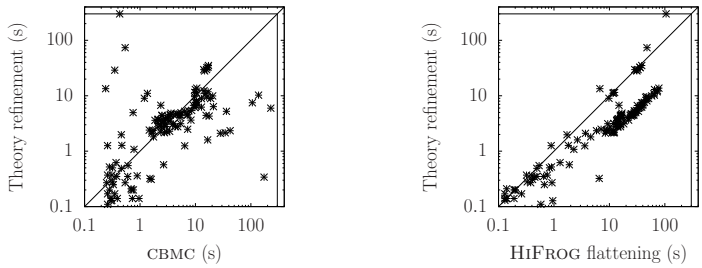

Fig. 4. Timings of CBMC (*left*) and HiFrog's flattening (*right*) against HiFrog's theory refinement for the safe instances.

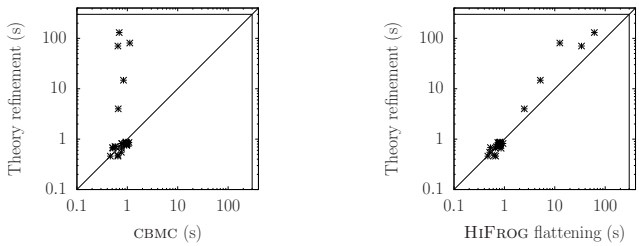

Fig. 5. Timings of CBMC (*left*) and HiFrog's flattening (*right*) against HiFrog's theory refinement for the unsafe instances.

indicating whether the bad behavior is reachable or not. Among safe instances, 17 require refinements.

For benchmarking we used Ubuntu 14.04 Linux system with two Intel Xeon E5620 CPUs clocked at 2.40GHz and 12 Gigabyte memory limit per process using a timeout of 300 seconds CPU time. The model checker was compiled with the GNU C++ compiler and the O3 optimization level. The complete experimental results, the source code, and a virtual machine are all available at [1].

Fig. 4 shows the verification results on safe properties. We compared (Fig. 4, *left*) the HiFrog's theory-refinement mode against CBMC version 5.7, the winner of the software model checking competition falsification track in 2017.[6] In 101 cases, HiFrog was either as fast or faster than CBMC, sometimes by orders of magnitude. Furthermore, HiFrog's theory refinement mode is compared against HiFrog's propositional flattening (Fig. 4, *right*), hence ensuring that the only difference in the solvers is in how the symbolic encoding is presented to the SMT solver. In 115 cases, the theory refinement was either as fast or faster than flattening in determining safety, providing a more convincing evidence that the theory refinement approach works well in practice.

[6] OpenSMT2: `https://scm.ti-edu.ch/repogit/opensmt2.git`, git ID: 99c960e4c; HiFrog (including CBMC that shares the CProver framework [2] with HiFrog): `https://scm.ti-edu.ch/repogit/hifrog`, git ID b35956f2c.

Table 2. Comparison of the heuristics against Min on instances requiring refinement.

	H0	H1	H2	H3	H4	H5	H6	H7	*Min*
#solved	**17**	16	**17**	**17**	**17**	**17**	**17**	**17**	17
#ref	660	2218	1250	1250	**533**	2266	1442	1831	162
time (s)	538	223	257	317	**123**	166	147	158	46.2

The verification results of unsafe benchmarks are shown in Fig. 5. In five cases, bug detection by HiFrog was slower than the one by CBMC since HiFrog required iterative refining of all the expressions to confirm the validity of the counter-example. However, in the remaining cases, HiFrog was comparable to CBMC.

6.1 Experiments on Refinement Heuristic

Alg. 1 does not address which exact statement should be refined based on a counter-example on Line 11 in case there are several possibilities. However this selection affects the run time of the model checking and is therefore of practical interest. We consider the following three features while building a refinement heuristic:

- Traversal order: the algorithm can proceed either by choosing from P the first statement (*forward order*) or the last statement (*backward order*) satisfying the condition on Line 11.
- All statements falsified by the counter-example are refined simultaneously (*simultaneous refinement*).
- All statements that depend on refined statements are refined simultaneously (*dependency refinement*).

The heuristics are as follows: H0 – Forward order; H1 – Backward order; H2 – Forward order with simultaneous refinement; H3 – Backward order with simultaneous refinement; H4 – Forward order with dependency refinement; H5 – Backward order with dependency refinement; H6 – Forward order with simultaneous and dependency refinement; and H7 – Backward order with simultaneous and dependency refinement. Based on the experimentation, the fastest solver on average results from using Forward order with dependency refinement. This is the heuristic we use in the results on Figs. 4-5. We briefly report on the results of the heuristics in Table 2 over the 17 instances of our total benchmark set where statements were refined. This benchmark set contains three crafted instances and the rest from the *bitvector* category of SV-COMP. The row labeled *#solved* reports how many instances the heuristic could solve before the timeout, *#ref* reports how many statements in total had to be refined over the set, and *time* reports the total run time. As a reference the table also reports results on the heuristic *Min* that requires no run time and computes a minimum set of refinements required to prove the property.

Finally, in Fig. 6 we show the reduction in the number of refined statements when using the *Min* heuristic on the 17 instances. As expected, the performance of the heuristic depends on the instance, but when effective, dramatically reduces the amount of flattened statements.

While the results are still preliminary mostly due to the prototype nature of the tools we are developing, we believe that they make a very strong point for the potential of the theory refinement approach in software model checking.

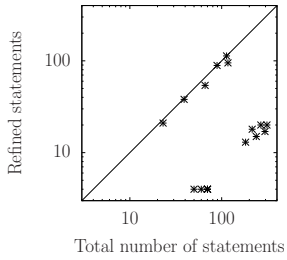

Fig. 6. The number of refined statements using the *Min* heuristic with respect to the total number of statements.

7 Conclusions and Future Work

We presented a new approach for abstraction refinement in software verification with SMT solvers. Our approach introduces iterative *theory refinement* and supports solving of formulas of combined theories in the SMT solver, where the binding to the theory is maintained by a series of identities in the original formula. Our main contribution is the gradual encoding process that uses the most precise theory only for a subset of all program statements, while handling the rest of the statements by using the less precise theories. This subset of the statements could either be identified by checking spurious counter-examples or simply specified by the user. Our framework can be extended by sets of theories with a partial order of refinement defined among them. In this paper, we demonstrated the framework on the UFP theory with the partial refinement to the BVP theory. We implemented this framework in the OpenSMT [19] solver and the model checker HiFrog [3].

We study different refinement strategies and compare them against a strategy computed off-line, as well as with the encoding into propositional logic, known as flattening or bit-blasting. Improvement is seen both in the running time and in the size of the resulting formula, demonstrating that the spurious counter-examples are usually eliminated by refining a small number of statements in the formula.

In future we plan to progress in several directions. We will study theory refinement with arithmetic theories and arrays, defining a partial order among theories based on the level of abstraction/refinement that they provide. We will further improve the automatic refinement based on an analysis of the counter-examples using approaches such as interpolation. We also plan to develop more sophisticated heuristics and strategies for refinement.

Acknowledgements. This work was supported by the SNF grants 163001 and 166288 and the SNF fellowship P2T1P2_161971.

References

1. http://verify.inf.usi.ch/hifrog/theoref
2. http://www.cprover.org/
3. 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 2017. LNCS, vol. 10205, pp. 207 – 213. Springer (2017)
4. Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: A proof-sensitive approach for small propositional interpolants. In: Proc. VSTTE 2015. LNCS, vol. 9593, pp. 1–18. Springer (2015)
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS 1999. LNCS, vol. 1579. Springer (1999)
6. Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer (2011)
7. Brady, B.A., Bryant, R.E., Seshia, S.A.: Learning conditional abstractions. In: Proc. FMCAD 2011. pp. 116–124. FMCAD Inc. (2011)
8. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. Journal on Satisfiability, Boolean Modeling and Computation 6, 165–201 (2009)
9. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Hanna, Z., Nadel, A., Palti, A., Sebastiani, R.: A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems. In: Proc. CAV 2007. LNCS, vol. 4590, pp. 547 – 560. Springer (2007)
10. Cimatti, A., Irfan, A., Griggio, A., Roveri, M., Sebastiani, R.: Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In: Proc. TACAS 2017. LNCS, vol. 10205, pp. 58 – 75 (2017)
11. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer (2000)
12. 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)
13. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F.: An efficient method of computing static single assignment form. In: Proc. POPL 1989. pp. 25–35. ACM (1989)
14. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
15. Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: Incremental upgrade checker for C. In: Proc. TACAS 2013. LNCS, vol. 7795, pp. 292–307. Springer (2013)
16. Gurfinkel, A., Belov, A., Marques-Silva, J.: Synthesizing safe bit-precise invariants. In: Proc. TACAS 2014. LNCS, vol. 8413, pp. 93–108. Springer (2014)
17. Hadarean, L., Bansal, K., Jovanović, D., Barret, C., Tinelli, C.: A tale of two solvers: Eager and lazy approaches to bit-vectors. In: Proc. CAV 2014. pp. 680 – 695. LNCS, Springer (2014)
18. Ho, Y.S., Chauhan, P., Roy, P., Mishchenko, A., Brayton, R.: Efficient uninterpreted function abstraction and refinement for word-level model checking. In: Proc. FMCAD 2016. pp. 65–72. ACM (2016)
19. 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)

20. Katz, G., Barrett, C., Harel, D.: Theory-aided model checking of concurrent transition systems. In: Proc. FMCAD 2015. pp. 81–88. IEEE (2015)
21. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View, Second Edition. Texts in Theoretical Computer Science. An EATCS Series, Springer (2016)
22. 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)
23. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)