Function Summarization Modulo Theories

Sepideh Asadi[1], Martin Blicha[14], Grigory Fedyukovich[2], Antti E. J. Hyvärinen[1], Karine Even-Mendoza[3], Natasha Sharygina[1], and Hana Chockler[3]

[1] Università della Svizzera italiana, Lugano, Switzerland `{firstname.lastname}@usi.ch`
[2] Princeton University, Princeton, USA, `grigoryf@cs.princeton.edu`
[3] King's College London, London, UK, `{firstname.lastname}@kcl.ac.uk`
[4] Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic

Abstract

SMT-based program verification can achieve high precision using bit-precise models or combinations of different theories. Often such approaches suffer from problems related to scalability due to the complexity of the underlying decision procedures. Precision is traded for performance by increasing the abstraction level of the model. As the level of abstraction increases, missing important details of the program model becomes problematic. In this paper we address this problem with an incremental verification approach that alternates precision of the program modules on demand. The idea is to model a program using the lightest possible (i.e., less expensive) theories that suffice to verify the desired property. To this end, we employ safe over-approximations for the program based on both function summaries and light-weight SMT theories. If during verification it turns out that the precision is too low, our approach lazily strengthens all affected summaries or the theory through an iterative refinement procedure. The resulting summarization framework provides a natural and light-weight approach for carrying information between different theories. An experimental evaluation with a bounded model checker for C on a wide range of benchmarks demonstrates that our approach scales well, often effortlessly solving instances where the state-of-the-art model checker CBMC runs out of time or memory.

1 Introduction

As programs become larger and more complex, they need more elaborated specifications to be verified for safety. Specifications with *multiple properties* are expensive to check as a significant amount of work is repeated over and over again. To overcome this issue, a verifier needs to operate incrementally. That is, the results obtained while verifying different properties should be reused to avoid wasting resources. When a specification involves certain amount of closely related properties, the incremental approaches are capable of avoiding verifying each property from scratch, and instead, they automatically identify and focus on small "deltas" in the verification conditions.

Verification approaches based on Satisfiability Modulo Theories (SMT) represent a program together with a specification in first-order logic. Often the specification is naturally expressible as a set of individual properties. Given a specification consisting of multiple properties, each property requires its own encoding that is precise enough to show the absence of spurious counterexamples to the property. In a real sense, this means that each formula requires its own *theory*: for example, some properties might be provable with a lightweight and inexpensive encoding, such as the one to the theory of equality and uninterpreted functions (EUF), while other properties might require expensive bit-precise reasoning. Identifying automatically which theory is suitable for verifying each property is challenging. In the incremental verification setting, maintaining such a framework gives new challenges. In this paper we solve this problem by designing the *theory interface* that enables migrating information among formulas in different theories. An over-approximating *function summary* [33, 3] is a well-known concept in Bounded Model Checking [11] that enables reuse of information among verification runs. Summaries are extracted using Craig interpolation [15] after a successful verification run for one property and used as a light-weight replacement of the precise encoding of the corresponding functions while verifying other properties. In this work, we propose an algorithm that effectively *incorporates different theories* for incremental verification of multiple properties via creation, reuse, and refinement of function summaries.

To the best of our knowledge this is the first integrated system for SMT-based, incremental model checking in which a sequence of safety properties is verified. Our algorithm works as follows. Given a program, a sequence of properties to verify, and an initially empty set of function summaries in several available theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$, the algorithm encodes the program and the current property using the least precise theory $\mathcal{T}_1$ and the least precise summaries available. In case the algorithm finds a proof, the result is sound since we guarantee that both the theories and the summaries always over-approximate the concrete program. Our algorithm starts with imprecise encodings since, if sufficient for proving a property, it lowers the cost of summarization and results in more compact summaries. If no proof is found, the algorithm increases the precision lazily. Assume that the problem is currently encoded using the theory $\mathcal{T}_i$. In the phase called *local refinement*, the algorithm sequentially adds summaries translated from theories $\mathcal{T}_j$, $j \neq i$ to $\mathcal{T}_i$ and checks if the property in this encoding is provable. The algorithm enters the second phase, *global refinement*, where the problem is encoded in a more precise theory $\mathcal{T}_{i+1}$, only when all summaries are already tried on theory $\mathcal{T}_i$. Then the algorithm returns to the local refinement again. Similarly to [33], our algorithm is capable of generating new function summaries and identifying actual bugs. Our refinement is driven by a counterexample-guided analysis that distinguishes spurious counterexamples from the real ones.

We have implemented the algorithm on top of the function-summarization-based bounded model checker HiFrog which uses the OpenSMT solver [24] for both solving and interpolation. Our implementation supports the theories of EUF, linear real arithmetic (LRA), and bit-vectors (BV). We provide an extensive evaluation on a range of large-scale benchmarks taken from SV-COMP[1] and crafted by ourselves. The tool exhibits a competitive performance compared to the state-of-the-art.

To sum up, our contributions are as follows:

- A novel approach to incremental verification that lazily identifies, among several suitable candidates, the lightest level of encoding for each given property.
- A theory interface for exchanging function summaries among formulas in different theories.
- An algorithm to leverage both function summaries and the overall precision of the program encoding that in practice demonstrates a competitive performance on a range of large-scale programs.

The rest of the paper is structured as follows. After a brief overview on related work in Sec. 2, Sec. 3 motivates the work with an example. Sec. 4 gives a background on function summarization and SMT, and Sec. 5 formally describes how the summary conversion is carried out between different SMT encodings. Sec. 6 presents our main algorithm for combining summary refinement and theory refinement. Finally, Sec. 7 reports on experimental evaluation, and Sec. 8 concludes the paper.

[1]Software Verification Competition, http://sv-comp.sosy-lab.org/, Linux Device Drivers (ldv) category.

2 Related Work

Our work is built on top of FunFrog [33], an approach for extracting and reusing interpolation-based function summaries in the context of Bounded Model Checking. The original work in FunFrog focuses only on propositional logic and does not consider the rich field of first-order theories available in modern SMT solvers. Consequently, despite behaving in an incremental manner, FunFrog is expensive in many cases in practice. In later work [3], we propose to use function summaries more generally by translating them to various SMT theories.

Different levels of abstraction were supported also by the previous version of our tool HiFrog [3]. However, information obtained from one level of abstraction can only be reused at the same level of abstraction. Our new approach has no such limitation and is able to convert and use information in form of function summaries obtained from the current level of encoding when working on different levels of encoding.

The idea of using an abstract description of the bit-precise level of encoding has been tried with success in verification of hardware designs [6] as well as software [19, 23]. The approaches use different level of encoding for different parts of the problem; these approaches typically start with uninterpreted functions and gradually refine to bit-level precision to rule out spurious counterexamples when necessary, while mixing different levels of encoding to verify a single property. Unlike these approaches, we do not mix different levels of encoding and only shift to more precise encoding globally, when the previous level of abstraction is insufficient. A single level of encoding allows us to extract useful information in form of function summaries from successful verification runs and reuse that information in the next verification run.

Both interpolants and function summaries are heavily used in model checking techniques. Interpolants are commonly used as a means of abstraction. Since McMillan's first application of interpolants in formal verification [27], interpolation has been applied in algorithms with various extensions in model checking [14, 28, 34, 21, 2, 32, 1, 29, 13, 35, 17, 16, 25]. Model checkers CPAchecker [10], SeaHorn [18], Ultimate Automizer [20] and others leverage interpolants in some form.

Function summaries date back to Hoare logic [22] where a pair of pre-condition and post-condition can be seen as an over-approximating function summary. Besides computation of function summaries using interpolation, function summaries have been computed using data-flow analysis [31, 8, 9] and iterative discovery of modification of variable values, used in model checkers Saturn [36] and Calysto [7]. However, all of these applications are orthogonal to our approach in incremental model checking.

3 Motivating Example

Fig. 1 shows a C program with a function call containing non-linear operations and two user-defined assertions. Our approach verifies the two assertions in the code incrementally. It is not hard to see that the program is safe with respect to both assertions. However, verification of this program using bit-precise encoding is expensive.

Our algorithm tries the less precise but easier to solve theory of equality and uninterpreted functions (EUF) as the level of abstraction first, leading to successful verification of the first assertion almost immediately. Moreover, it generates and stores a summary for function `func`. To verify the second assertion, reasoning over linear real arithmetic (LRA) is necessary. Our algorithm presented later in the paper enables to translate the summary for `func` from EUF to LRA and to reuse it to successfully verify the second assertion.

```
int a, b, c;

void func() {
  c = b;
  if (a > 0) a = b;
  int m = 0;
  for (int i = 0; i < 100; i++)
    m += a*b;
  b = m;
}
```

```
int main() {
  a = nondet(); b = nondet();
  if (a <= 0) return -1;
  func();
  assert(a == c);
  if (a > 0) {
    func();
    if (c > 10) assert(a > 7);
  }
  return 0;
}
```

Figure 1: Program in C with non-linear arithmetic.

4 Background and Previous Work

Our discussion relies heavily on concepts used in SMT solving. In the following, we will define the notation that we will use in the presentation. A *signature* Σ is a union of $\mathcal{C}, \mathcal{F}, \mathcal{P}$, pairwise disjoint sets of constants, and function and predicate symbols respectively. Each function in $\mathcal{F}$ is associated with an arity $n \geq 1$.

As usual, a set of *terms* $\mathcal{R}_\Sigma$ is defined inductively comprising constants, variables, and more complex expressions by applying function symbols on terms. The rules are captured by the following grammar:

$$\begin{aligned} \textit{term} ::= {} & \textit{const} \\ & \mid \textit{var} \\ & \mid f(\textit{term}, \ldots, \textit{term}) \end{aligned}$$

where *const* $\in \mathcal{C}$ is a constant, *var* is a variable, and $f \in \mathcal{F}$ is a function symbol with arity equal to the number of terms in parentheses. Similarly, a set of *formulas* $\mathcal{S}_\Sigma$ is built inductively using the following grammar:

$$\begin{aligned} \textit{formula} ::= {} & \textit{Bvar} \\ & \mid p(\textit{term}, \ldots, \textit{term}) \\ & \mid \textit{term} = \textit{term} \mid \top \mid \bot \mid \neg \textit{formula} \\ & \mid \textit{formula} \wedge \textit{formula} \mid \textit{formula} \vee \textit{formula} \end{aligned}$$

where *Bvar* is a Boolean variable, $p \in \mathcal{P}$ is a predicate symbol with arity equaling to the number of terms in parentheses, and $\top$ and $\bot$ are Boolean constants denoting *true* and *false* respectively. Finally, a quantifier-free first-order theory $\mathcal{T}_\Sigma \subseteq \mathcal{S}_\Sigma$ is a set of formulas defined over the signature Σ. When $\mathcal{T}$ is clear from the context, we call a formula from $\mathcal{S}_\Sigma$ a *Satisfiability modulo theory (SMT) instance.*

4.1 Programs and Summaries

A *loop-free program* is a tuple $P = (F, \textit{main})$, such that F is a finite set of non-recursive functions, and *main* $\in F$ is an entry point. Let set $\hat{F}$ gather all function calls from F, where $\hat{f}$ is a call of function f. In $\hat{F}$ we distinguish different calls to the same function f by enumerating them as $\hat{f}_1, \ldots, \hat{f}_n$. A *summary* of a function f is a relation over the input and output variables

```
// encoding of the first call of function ‘‘func’’:
```

$$c_1 = b_0 \wedge a_1 = ite(a_0 > 0, b_0, a_0) \wedge m_0 = 0 \wedge \text{L\_UNW}_1 \wedge b_1 = m_{10}$$

```
// encoding of the second call of function ‘‘func’’:
```

$$c_2 = b_1 \wedge a_2 = ite(a_1 > 0, b_1, a_1) \wedge m_{11} = 0 \wedge \text{L\_UNW}_2 \wedge b_2 = m_{21}$$

```
// encoding of function ‘‘main’’:
```

$$a_0 > 0 \wedge a_1 > 0 \wedge c_2 > 10$$

```
// encoding of the negation of the first assertion:
```

$$\neg(a_1 = c_1)$$

```
// encoding of the negation of the second assertion:
```

$$\neg(a_2 > 7)$$

Figure 2: Modular encoding of program from Fig. 1 to an SMT formula.

of f that over-approximates the precise behavior of f. That is, if a formula $f_{precise}$ encodes the body of f, and f_{sum} encodes its summary, then $f_{precise} \implies f_{sum}$ must hold.

In this work, we reduce a Bounded Model Checking (BMC) [11] task to an SMT task. That is, a program is encoded to a quantifier-free first-order formula in a given theory $\mathcal{T}$, which is then solved for satisfiability. Our intent is to allow function calls in the considered programs and to over-approximate them by summaries whenever applicable. If the program encoding is inconsistent with the negation of safety specification, then the program is safe.

We construct summaries using Craig Interpolation [15], a widely used technique to create over-approximations in Model Checking. Given a pair of formulas (A, B), *Craig interpolant* of (A, B) is a formula I such that $A \implies I$, $I \wedge B$ is unsatisfiable, and I defined over symbols appearing both in A and B. In our context, unsatisfiable formulas originate from bug-free programs, and thus the summaries express that no trace allowed by the function body leads to a violation of the considered safety specification. In order to construct and use function summaries in the context of BMC, we assume that a BMC formula is a conjunction of encodings of individual function calls. Thus, the problem of determining whether the program is safe with respect to a safety assertion Q reduces to the problem of determining the satisfiability of the SMT formula

$$\bigwedge_{\hat{f} \in \hat{F}} \text{ENCODE}(\hat{f}) \wedge \neg\text{ENCODE}(Q) \implies \bot.$$

We illustrate the encoding on the following example.

Example 1. *Fig. 2 shows a (simplified) encoding of program from Fig. 1 to an SMT formula. The formula consists of five parts: a conjunct representing function* `main`*, two equivalent (modulo renaming) conjuncts representing calls of* `func`*, and two conjuncts representing the negated assertions. As customary in BMC, each program variable has its indexed copies (induced by the single static assignment form). The formulas* L_UNW_i*,* $i \in \{1, 2\}$*, represent a loop unwinding. Note that the encoding of* `main` *consists only of the path condition to assertions, but in general it should explicitly encode all possible paths through the function body.*

In [33], we presented a method to extract summaries for every function call $\hat{f}$ exploiting the proof of unsatisfiability of this formula. In a nutshell, the approach considers a conjunction of the encoding of all nested function calls from $\hat{f}$, i.e., $f_{precise} \overset{\text{def}}{=} \text{ENCODE}(\hat{f}) \wedge$

$\bigwedge_{\hat{g} \in nested\_calls(\hat{f})}$ ENCODE$(\hat{g})$, treats it as A, treats the rest of the program encoding (including the negation of assertion) as B, and interpolates. Note that the resulting interpolant f_{sum} can now be used in place of $f_{precise}$ when creating the formula again because by construction $f_{precise} \implies f_{sum}$.

Example 2. *A possible function summary for* `func` *obtained after verifying the first assertion is* $(a_0 > 0) \implies (a_1 = c_1)$*. It can successfully replace both calls to* `func` *(after the variables are renamed to match the second call) while verifying the second assertion.*

Note that for the examples above, using two SMT precisions are enough: EUF for the first assertion, and LRA for the second one.

5 Theory-Based Model Refinement

This section presents a general framework that allows a translation back and forth among theories of SMT with different level of precision.

Our work views the problem of bounded model checking of C programs as a decision problem which is (i) decidable, and (ii) not based on Nelson-Oppen theory combination [30]. We may therefore concentrate in our framework on four theories of interest: the quantifier-free theories of equality and uninterpreted functions (EUF), linear real arithmetic (LRA), non-linear real arithmetic (NRA), and bit-vectors (BV)[2]. As a result, we obtain a decision procedure that has a relatively low complexity. Our framework, called *theory interface*, provides a common place from which the theories are instantiated, and to which they can also be converted back. This theory interface is not aimed to be passed to an SMT solver, but instead provides an infrastructure through which an instance from one theory can be converted to an instance from another theory.

The transformation from a theory $\mathcal{T}$ to the theory interface and back can be expressed in the theory-specific instantiations of the following rules, where $[\phi]^{\mathcal{T}}$ denotes that the expression ϕ is encoded using theory $\mathcal{T}$:

$$\frac{[f(\mathbf{t})]^{\mathcal{T}}}{f([\mathbf{t}]^{\mathcal{T}})} \quad \text{if } f([\mathbf{t}]^{\mathcal{T}}) \text{ in } \mathcal{T} \qquad\qquad \frac{[f(\mathbf{t})]^{\mathcal{T}}}{v_{f(\mathbf{t})}} \quad \text{if } f([\mathbf{t}]^{\mathcal{T}}) \text{ not in } \mathcal{T} \tag{1}$$

We use the notation $f(\mathbf{t})$ to abbreviate $f(t_1, \ldots, t_n)$, and $f([\mathbf{t}]^{\mathcal{T}})$ to abbreviate $f([t_1]^{\mathcal{T}}, \ldots, [t_n]^{\mathcal{T}})$. Above we write $f([\mathbf{t}]^{\mathcal{T}})$ *in* $\mathcal{T}$, if $f \in \Sigma$ and there is a derivation recursively using the rules (1) such that $f([\mathbf{t}]^{\mathcal{T}})$ is expressible in $\mathcal{T}$. We denote by $v_{f(\mathbf{t})}$ a variable that is unique to the expression $f(\mathbf{t})$. For example, the expression $f(x, x)$ is not expressible in the theory of linear real arithmetic if f is the multiplication operation and x is a variable, and therefore the result of applying the rules (1) is v_{x*x}. To simplify slightly the notation, we define a bijection $\mathcal{M}$ that maps terms $f(\mathbf{t})$ to the variables $v_{f(\mathbf{t})}$. For completeness, we present the three rules for transforming non-atomic formulas

$$\frac{[\phi_1 \wedge \phi_2]^{\mathcal{T}}}{[\phi_1]^{\mathcal{T}} \wedge [\phi_2]^{\mathcal{T}}} \qquad \frac{[\phi_1 \vee \phi_2]^{\mathcal{T}}}{[\phi_1]^{\mathcal{T}} \vee [\phi_2]^{\mathcal{T}}} \qquad \frac{[\neg\phi]^{\mathcal{T}}}{\neg[\phi]^{\mathcal{T}}} \tag{2}$$

that are independent of a theory and thus common to all transformations.

[2]For the signature of bit-vectors, we use a modification presented in [23] that preserves the high-level programming language structures to facilitate the proofs of over-approximation.

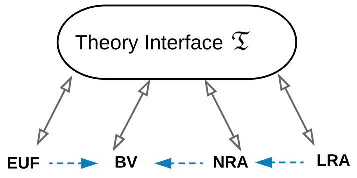

Figure 3: Theory interface between EUF, LRA, NRA, and BV. The horizontal arrows demonstrate the relation among these theories from the perspective of over-approximation. This relation is a part of the contribution of this work.

5.1 Theory Interface

A *theory interface* $\mathfrak{T}$ is a general representation for formulas that we use for transformation among theories. Fig. 3 outlines a communication among our four theories of interest. Because this paper aims at using from early on a light-weight theory that suffices for reasoning, over-approximation among theories is at the core of speeding up the solving procedure. In the rest of this section, we formally define a theory interface and establish a relation among theories in a sound way.

Definition 1 (Theory interface $\mathfrak{T}$)**.** *Given a sequence of theories $\mathcal{T}_1, \dots, \mathcal{T}_n$ with signatures $\Sigma_1, \dots, \Sigma_n$ respectively, a theory interface $\mathfrak{T}$ is a tuple $\langle \Sigma, \mathcal{M}_1, \dots, \mathcal{M}_n \rangle$ where $\Sigma \stackrel{\text{def}}{=} \Sigma_1 \cup \dots \cup \Sigma_n$, and each $\mathcal{M}_i$ is a* bijective *mapping $\mathcal{M}_i : (\mathcal{S}_\Sigma \cup \mathcal{R}_\Sigma) \setminus (\mathcal{S}_{\Sigma_i} \cup \mathcal{R}_{\Sigma_i}) \to \mathcal{X}_i$ where $\{\mathcal{X}_i\}_{0<i\leq n}$ are pairwise disjoint sets of unique variables not used anywhere else.*

Intuitively, $\mathcal{M}_i$ replaces the formulas and terms that are not expressible in theory $\mathcal{T}_i$ by unique fresh variables. Note that for every $\mathcal{T}_i$, $\mathcal{S}_{\Sigma_i} \subseteq \mathcal{S}_\Sigma$ and $\mathcal{R}_{\Sigma_i} \subseteq \mathcal{R}_\Sigma$.

The projection of $\mathfrak{T}$ to one of the theories $\mathcal{T}_i$ is done by the following rules. First, if $f(\mathbf{t}) \in \mathcal{S}_{\Sigma_i}$ (i.e., is expressible in theory $\mathcal{T}_i$), then it is projected to $\mathcal{T}_i$ without changes. Second, if $f(\mathbf{t}) \notin \mathcal{S}_{\Sigma_i}$ (i.e., is not expressible in theory $\mathcal{T}_i$), then we replace it by a fresh symbol $\mathcal{M}_i(f(\mathbf{t})) \stackrel{\text{def}}{=} v_{f(\mathbf{t})} \in \mathcal{X}_i$. For transformation in the opposite direction, i.e., $\mathcal{T}_i$ to $\mathfrak{T}$, we define the inverse function $\mathcal{M}_i^{-1}$ as $\mathcal{M}_i^{-1} : v_{f(\mathbf{t})} \mapsto f(\mathbf{t})$ for $v_{f(\mathbf{t})}$ in the range of $\mathcal{M}_i$.

In the following, we develop a set of translation functions to different theories and build the over-approximation relation among these translation functions. Given a formula ϕ in theory interface $\mathfrak{T}$ and an arbitrary theory $\mathcal{T}$, we write $Tr_{\mathcal{T}}(\phi)$ for the translation from ϕ to $\mathcal{T}$.

Definition 2 (over-approximation)**.** *Let ϕ be a formula in $\mathfrak{T}$, and $\mathcal{T}_1$ and $\mathcal{T}_2$ two arbitrary theories. The two translation functions, $Tr_{\mathcal{T}_1}(\phi)$ and $Tr_{\mathcal{T}_2}(\phi)$ convert the original formula ϕ into $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. We say that $\mathcal{T}_1$ over-approximates $\mathcal{T}_2$ if*

$$Tr_{\mathcal{T}_1}(\phi) \models \bot \textit{ implies } Tr_{\mathcal{T}_2}(\phi) \models \bot \tag{3}$$

We give the specifics for the theories EUF and LRA, but in the interest of space, provide the rules of transformation from theory interface to BV and NRA in Appendix A. To establish the over-approximation relation, we assume in this paper that the programs being verified admit no overflows or underflows, and that their semantics can be exactly captured by BV.

Definition 3 (Theory of EUF)**.** *Let $\mathcal{X}$ be a set of variables and $\mathcal{F}$ be a set of function symbols with arities. An Equality logic formula with uninterpreted functions (*EUF*) is defined by the* *grammar*

$$
\begin{aligned}
trm &::= const \\
&\quad \mid var \\
&\quad \mid f(trm, \ldots, trm) \qquad \textit{where } f \textit{ is uninterpreted} \\
fla &::= Bvar \\
&\quad \mid p(trm, \ldots, trm) \qquad \textit{where } p \textit{ is uninterpreted} \\
&\quad \mid trm = trm \mid trm \neq trm \mid \top \mid \bot \mid \neg fla \\
&\quad \mid fla \wedge fla \mid fla \vee fla \mid
\end{aligned}
$$

where fla is a quantifier-free formula, $var \in \mathcal{X}$, $f \in \mathcal{F}$, and $const \in \mathcal{C}$. With the exception of equality and disequality ($=, \neq$), function and predicate symbols are treated as uninterpreted.

Semantically, EUF has the axioms of reflexivity, symmetry and transitivity for the symbol of equality, and congruence axiom for function and predicate symbols $(\mathbf{x} = \mathbf{y}) \rightarrow (f(\mathbf{x}) = f(\mathbf{y}))$ and $(\mathbf{x} = \mathbf{y}) \rightarrow (p(\mathbf{x}) \leftrightarrow p(\mathbf{y}))$ where $\mathbf{x} = \mathbf{y}$ is an abbreviation for $(x_1 = y_1) \wedge \ldots \wedge (x_n = y_n)$ and f and p are function and predicate symbols, respectively, of arity n.

Definition 4. *A quantifier-free formula in the language of theory of Linear Real Arithmetic (LRA) is defined by the following grammar:*

$$
\begin{aligned}
trm &::= const \\
&\quad \mid var \\
&\quad \mid const * var \\
&\quad \mid f(trm, \ldots, trm) \qquad \textit{where } f \in \{+\} \\
fla &::= Bvar \\
&\quad \mid p(trm, \ldots, trm) \qquad \textit{where } p \in \{\leq, <\} \\
&\quad \mid \top \mid \bot \mid \neg fla \\
&\quad \mid fla \wedge fla \mid fla \vee fla \mid
\end{aligned}
$$

where var are variables, and const is a rational number.

5.2 Encoding of Theory Interface into Specific Theories

Light-weight theories help removing overly complex or irrelevant details from the encoding of a program whenever possible. We define the following rules for the theory-specific part of the transformation from $\mathfrak{T}$ to EUF:

$$
\begin{gathered}
\frac{[v]^{EUF}}{v} \; v \text{ is a variable or a constant} \\[2ex]
\frac{[t_1 = t_2]^{EUF}}{[t_1]^{EUF} = [t_2]^{EUF}} \\[2ex]
\frac{[t_1 \bowtie t_2]^{EUF}}{(\neg [t_1 = t_2]^{EUF}) \wedge ([t_1]^{EUF} \bowtie [t_2]^{EUF})} \; \bowtie \in \{>, <\} \\[2ex]
\frac{[f(\mathbf{t})]^{EUF}}{f([\mathbf{t}]^{EUF})} \; \text{otherwise}
\end{gathered}
\tag{4}
$$

Note that in the third rule of (4), if the function symbol $>$ or $<$ is applied over the terms of theory interface, it can be simply translated into a disequality in EUF. All the other cases in the signature of theory interface which cannot be applied in the first three rules such as $\{\leq, \geq, \ldots\}$ are handled by the fourth rule.

Theorem 1. *For every $\phi \in \mathfrak{T}$, $Tr_{EUF}(\phi) \models \bot$ implies $Tr_{BV}(\phi) \models \bot$.*

Proof. We show that every model in BV can be translated to a model in EUF.

Assume that there is a satisfying assignment in BV, such that $a = b$ holds for two bitvectors a and b. This can be trivially translated to an equality $a = b$ in EUF.

In case of equality of two function applications $f(a) = f(b)$, we utilize the congruence rule in EUF, assuming that each function in BV is implemented as a deterministic circuit. $\square$

We define the following rules to transform $\mathfrak{T}$ to LRA[3]:

$$\left.\begin{array}{lr}
\dfrac{[t_1 = t_2]^{LRA}}{([t_1]^{LRA} \leq [t_2]^{LRA}) \wedge ([t_2]^{LRA} \leq [t_1]^{LRA})} & (5.1) \\[2ex]
\dfrac{[v]^{LRA}}{v} \quad v \text{ is a variable or an integer constant} & (5.2) \\[2ex]
\dfrac{[t_1 + t_2]^{LRA}}{[t_1]^{LRA} + [t_2]^{LRA}} & (5.3) \\[2ex]
\dfrac{[-t_1]^{LRA}}{(-1) * [t_1]^{LRA}} & (5.4) \\[2ex]
\dfrac{[t_1 * t_2]^{LRA}}{[t_1]^{LRA} * [t_2]^{LRA}} \quad t_1 \text{ or } t_2 \text{ is an integer constant} & (5.5) \\[2ex]
\dfrac{[f(\mathbf{t})]^{LRA}}{\mathcal{M}(f(\mathbf{t}))} \text{ otherwise} & (5.6)
\end{array}\right. \qquad (5)$$

The rule (5.6) uniquely associates the expression with a fresh variable. Essentially this rule is used for over-approximation of all the expressions that cannot be expressed in sufficient precision in LRA. The rules (5.2) and (5.5) operate only on integer constants in order to preserve the soundness of translation between LRA and BV. Example 3 illustrates this case in detail.

Example 3 (**Over-approximation of** BV **by** LRA)**.** *Consider the following excerpt of a program written in C:* `int x = 1; int y = 0.5 * x; assert ( y == 0 );` *Let $\phi \overset{\text{def}}{=} x = 1 \wedge 0.5 * x = 0$ represent the corresponding SMT representation. Following the semantics of C, a bit-precise encoding of ϕ is satisfiable since $0.5*1$ is truncated to 0. However, an* LRA*-encoding of ϕ is unsatisfiable. According to Def. 2, this means that* LRA *does* not *over-approximate* BV*. In order to get that over-approximating behavior, we impose restrictions on* LRA *rules (5.2) and (5.5) and apply rule (5.6) when these restrictions are not met. The translation applied to ϕ results in $x = 1 \wedge v_{0.5*x} = 0$ which is satisfiable in* LRA*. The same restrictions are imposed in* NRA *(see Appendix A).*

[3] We assume that before undergoing a transformation, a preprocessing is done for the sake of normalization, e.g., $-1 * 2 * x$ is normalized to $-2 * x$.

Theorem 2. *For every $\phi \in \mathfrak{T}$, $Tr_{LRA}(\phi) \models \bot$ implies $Tr_{BV}(\phi) \models \bot$.*

Proof. We show that every model in BV can be translated to a model in LRA. Assume that there are no overflows or underflows in BV. This guarantees that the models of all arithmetic operations in BV are also models in LRA. □

5.3 Decoding Theories to the Theory Interface

The previous section describes the instantiation from the theory interface to a specific theory of interest. This section presents the inverse, that is, transforming from a theory to the theory interface. Such steps are necessary in order to build the over-approximation relation among theories. The key insight is to use the mapping $\mathcal{M}^{-1}$. The transformation from EUF to $\mathfrak{T}$ is defined by the following rules:

$$\begin{gathered}\frac{[t_1 = t_2]^{\mathfrak{T}}}{[t_1]^{\mathfrak{T}} = [t_2]^{\mathfrak{T}}} \\ \frac{[v]^{\mathfrak{T}}}{v}\ v \text{ is a variable or a constant} \\ \frac{[f(\mathbf{t})]^{\mathfrak{T}}}{f([\mathbf{t}]^{\mathfrak{T}})}\end{gathered} \tag{6}$$

Similarly, the rules for transforming from LRA to theory interface $\mathfrak{T}$ are as follows:

$$\begin{gathered}\frac{([t_1]^{\mathfrak{T}} \leq [t_2]^{\mathfrak{T}}) \wedge ([t_2]^{\mathfrak{T}} \leq [t_1]^{\mathfrak{T}})]}{[t_1]^{\mathfrak{T}} = [t_2]^{\mathfrak{T}}} \\ \frac{[t_1 \bowtie t_2]^{\mathfrak{T}}}{[t_1]^{\mathfrak{T}} \bowtie [t_2]^{\mathfrak{T}}}\ \text{ is a function or predicate symbol in } LRA \\ \frac{[v]^{\mathfrak{T}}}{v}\ v \text{ is a variable or a constant}, v \notin dom(\mathcal{M}^{-1}) \\ \frac{[v]^{\mathfrak{T}}}{\mathcal{M}^{-1}(v)}\ v \in dom(\mathcal{M}^{-1})\end{gathered} \tag{7}$$

Determining satisfiability in an over-approximative theory does not guarantee that the formula is satisfiable in a more precise theory, since the satisfiability might have been introduced by the abstraction. In such cases the strength of the formula must be enhanced through techniques such as refinement. In the next section, we discuss how to use the theory-based model refinement idea in a model-checking algorithm.

6 Summary and Theory-Aware Model Checking

Our novel approach to incremental bounded model checking is presented in Alg. 1. It takes as input a program with a sequence $\langle Q_1, \ldots, Q_m \rangle$ of safety assertions that are to be verified,

```
Algorithm 1: VERIFY($P, \langle \mathcal{T}_1, \dots, \mathcal{T}_n \rangle, \langle Q_1, \dots, Q_m \rangle$)
  Input: Program $P$ with function calls $\hat{F}$, sequence of theories $\langle \mathcal{T}_1, \dots, \mathcal{T}_n \rangle$; sequence of
         safety assertions $\langle Q_1, \dots, Q_m \rangle$
  Output: Verification result: {SAFE, UNSAFE}
1 for each $\mathcal{T}_j$ do
2     for each $\hat{f} \in \hat{F}$ do $\sigma_{\mathcal{T}_j}(\hat{f}) \leftarrow true$;
3 for each $Q_i$ do
4     for each $\mathcal{T}_j$ do
5         $\langle result, \sigma_{\mathcal{T}_j} \rangle \leftarrow$ SUMREF($P, \mathcal{T}_j, \langle \sigma_{\mathcal{T}_1}, \dots, \sigma_{\mathcal{T}_n} \rangle, Q_i$);
6         if $result =$ SAFE then break;
7         if $j = n$ then return UNSAFE;
8 return SAFE;
```

```
Algorithm 2: SUMREF($P, \mathcal{T}, \langle \sigma_{\mathcal{T}_1}, \dots, \sigma_{\mathcal{T}_n} \rangle, Q$)
  Input: Program $P = (F, f_{main})$ with function calls $\hat{F}$, theory $\mathcal{T}$; sequence $\langle \sigma_{\mathcal{T}_1}, \dots, \sigma_{\mathcal{T}_n} \rangle$
         of mappings of function calls to their summaries; $Q$: safety assertion to verify
  Output: Verification result: {SAFE, UNSAFE}, updated $\sigma_{\mathcal{T}}$
  Data: $\varphi$: BMC formula, $WL \subseteq \hat{F}$, $Pr$: precision mapping for function calls, $CE$:
        counterexample
1 $\varphi \leftarrow$ ENCODE$_{\mathcal{T}}(\hat{main}) \land \neg$ENCODE$_{\mathcal{T}}(Q)$;
2 for each $\hat{f} \in \hat{F}$ do $Pr(\hat{f}) \leftarrow 0$;
3 while $true$ do
4     $\langle result, CE \rangle \leftarrow$ SOLVE($\varphi$);
5     if $result =$ SAT then
6         $WL \leftarrow$ GETCALLSWITHWEAKSUMMS($CE$);
7         if $WL = \varnothing$ then return UNSAFE;
8         for each $\hat{f} \in WL$ do
9             if $Pr(\hat{f}) < n$ then
10                $Pr(\hat{f}) \leftarrow Pr(\hat{f}) + 1$;
11                $\psi \leftarrow \sigma_{\mathcal{T}_{Pr(\hat{f})}}(\hat{f})$;
12                $\varphi \leftarrow \varphi \land$ TRANSLATE$_{\mathcal{T}}(\psi)$;
13            else
14                $\varphi \leftarrow \varphi \land$ ENCODE$_{\mathcal{T}}(\hat{f})$;
15    else
16        for each $\hat{f} \in \hat{F}$ do
17            $\sigma_{\mathcal{T}}(\hat{f}) \leftarrow \sigma_{\mathcal{T}}(\hat{f}) \land$ GETITP$_{\mathcal{T}}(\varphi, \hat{f})$;
18        return $\langle$SAFE, $\sigma_{\mathcal{T}} \rangle$;
```

and a sequence of theories $\langle \mathcal{T}_1, \dots, \mathcal{T}_n \rangle$, such that for each i and j, $i < j$, $\mathcal{T}_j$ is *not* an over-approximation of $\mathcal{T}_j$.[4] For simplicity, we assume that all assertions are located in the entry function (i.e., $\hat{main}$), but our implementation does not have this restriction. We refer to $\sigma_{\mathcal{T}_j}(\hat{f})$

[4]In our implementation, we chose $\mathcal{T}_1 =$ EUF, $\mathcal{T}_2 =$ LRA, and $\mathcal{T}_3 =$ BV.

as to a summary for function f which is encoded in theory $\mathcal{T}_j$. Note that the function summary is initialized with the weakest possible summary, namely *true*. The algorithm searches for a first assertion which does not hold and then terminates with the UNSAFE result. If no such assertion is found, the algorithm terminates with the SAFE result.

Alg. 1 maintains a set of mappings for each function call and each theory to a summary formula that over-approximates the behavior of the source function and is expressible in the theory. These summary formulas are initially true, but are refined after a verification run of each assertion Q_i. Importantly, they are reused by a verification run of the next assertion Q_{i+1}.

An algorithm for verifying an assertion Q with function summaries is shown in Alg. 2. It starts by encoding the entry function in a given theory $\mathcal{T}$ and conjoins it with the negation of encoding of Q in $\mathcal{T}$. If this formula φ is unsatisfiable, then Q holds, manifesting the weakest possible summary *true* was adequate for all nested function calls from $\hat{main}$. Otherwise, our algorithm starts gradually strengthening the formula φ by adding summaries of the function calls responsible for the satisfiability of φ. We rely on a method described in [33] to get models of satisfiable formulas and identifying the "reason" for their satisfiability.

Our new contribution is a method to refine summaries based on lazy enumeration of available theories. In particular, Alg. 2 maintains a level of precision for each function call. In each round of refinement, if a function call $\hat{f}$ requires strengthening, its level of precision is increased by one, and a summary of that level, if available, is conjoined to φ. The key ingredient here is the set of translation rules, described in the previous section, that allow effectively reusing formulas among theories. Note that the translation process is not direct, but operates via a theory interface (omitted from the pseudocode in order to save space).

In order to prove the soundness of Alg. 1, we need to show that a summary in one theory can be reused in another theory. In other words, the correctness of Alg. 1 depends on the correctness of transferral of summaries from one theory to another theory. To this end, we connect the over-approximations via function summarization with the over-approximations via less precise theory. The following theorem captures formally the correctness of summary transformation through theory interface.

Theorem 3. *Let f be a function, $f^{\mathcal{T}}_{sum}$ be a summary of f obtained from $f^{\mathcal{T}}_{precise}$, and $f^{\mathcal{T}'}_{sum}$ be a translation of $f^{\mathcal{T}}_{sum}$ to theory $\mathcal{T}'$. Then $f^{\mathcal{T}'}_{sum}$ is also a summary of f.*

Proof. First, notice that by translating back $f^{\mathcal{T}}_{sum}$ to the theory interface using the rules in (1) we obtain an over-approximating representation f_{sum} of $f_{precise}$. This follows from the properties of the translation. Next, by translating f_{sum} to theory $\mathcal{T}'$ using rules in (1) we obtain an over-approximating formula $f^{\mathcal{T}'}_{sum}$ of f_{sum}. Finally, by transitivity $f^{\mathcal{T}'}_{sum}$ over-approximates $f_{precise}$, and hence $f^{\mathcal{T}'}_{sum}$ is a summary of f as stated in the theorem. □

Note that the fact that $f^{\mathcal{T}'}_{sum}$ is a summary of f is sufficient for correctness of using $f^{\mathcal{T}'}_{sum}$ instead of $f^{\mathcal{T}'}_{precise}$ in next verification tasks in case of unsatisfiability results. It is not required that $f^{\mathcal{T}'}_{sum}$ over-approximates $f^{\mathcal{T}'}_{precise}$. In case of over-approximating theory $\mathcal{T}'$ it may happen that the full encoding of a function f, $f^{\mathcal{T}'}_{precise}$, is not sufficient to prove a property while a summary obtained from a different theory might be enough.

7 Implementation and Evaluation

We have implemented our summary and theory refinement algorithm on top of HiFrog, an incremental bounded model checker. On the backend, HiFrog uses the SMT solver OpenSMT [24] which is equipped with a flexible interpolation framework for EUF [5] and LRA [4] for computing function summaries. Technical information about the setup of the tool and evaluation results can be found at http://verify.inf.usi.ch/sum-theoref.

With the reported experiments, our goal is to understand how bounded model checking can benefit from using over-approximative techniques based on function summaries obtained from SMT theories. We therefore compare our implementation against CBMC v5.8 [26], the most efficient bounded model checker based on the results of Competition on Software Verification SV-COMP. Compared to an earlier version of HiFrog [3], this work 1) automates the theory refinement which previously required manual intervention and 2) transfers the summaries among theories, which previously was not supported at all. In the following, we also give an explicit experimental comparison against our earlier version to highlight the usefulness of the proposed algorithm.

We instantiated the summary and theory refinement framework as described by Alg. 1 and Alg. 2 with three theories: EUF, LRA and BV (using a standard encoding to propositional logic). In the global refinement phase of Alg. 1, the program is first encoded in EUF. In case of unsuccessful verification with EUF, the entire program is encoded in LRA. Where the verification with LRA fails, the entire program falls back on bit-blasting. In the local refinement phase, in each of these stages, summaries of functions are used when available and are refined on demand. After a successful verification run, summaries are extracted in the current theory and become available for verification of the subsequent assertions. Using the framework described in Sec. 5, they are translated to different theories on-demand.

Currently in our implementation, only EUF and LRA theories may exchange summaries. However, before the precise bit-blasting of the entire program, we can bit-blast the more abstract EUF and LRA summaries. While this feature is currently under development, we believe that it will lead to smaller and more compact proofs and thus improve efficiency of the entire tool. Similarly, the inverse direction of extracting high-level information from bit-precise summaries remains a future work.

7.1 Results

For benchmarking we used an Ubuntu 16.04 Linux system with two Intel Xeon E5620 CPUs clocked at 2.40GHz. We limit the memory consumption to 2 Gigabytes and the CPU time to 200 seconds per process.

We chose 109 C programs from the `ldv` category of SV-COMP that either CBMC or HiFrog could solve withn our time and memory limits. Our choice of the `ldv` benchmarks is justified because they exercise our algorithm in an interesting way due to containing many assertions and functions, and being relatively large. We excluded programs where CBMC reported an internal error. In addition, we included 31 tricky hand-crafted smaller programs to stress-test our implementation. On average, the benchmarks have 10'000 lines of code, the longest ones reaching to 35'000 lines of code.

In total, our benchmark set contains 140 C programs and 500 assertions (verification tasks) placed inside these programs. 215 of these assertions were recognized as unreachable statements by the entry function in the C program. We excluded them from our study and focused on those tasks that require a full solving procedure. This narrowed down our set to 285 verification tasks.

In the following, we provide more details on statistics we collected after the extensive evaluation of our algorithm against CBMC and three individual verification approaches from the older version of HiFrog, namely pure EUF, LRA, BV. Table 1 gives statistics on our benchmark

Table 1: HiFrog against cbmc, and the original version of HiFrog with respect to pure EUF, LRA, and BV solving, where **#sv** is the number of benchmarks from SV-COMP, and **#craft** is the number of our tricky hand-crafted benchmarks.

Tools	**Solved**		**Timeouts**		**Memory outs**		**Unknown**	
	#sv	#craft	#sv	craft	#sv	#craft	#sv	#craft
HiFrog	**67**	**31**	32	**0**	**10**	0	-	-
CBMC	63	11	**28**	20	18	0	-	-
EUF only	49	0	38	0	**10**	0	12	31
LRA only	48	1	40	0	11	0	10	30
BV only	43	4	33	4	33	23	-	-

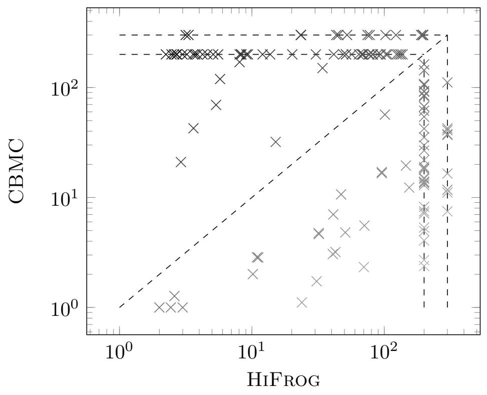

Figure 4: HiFrog vs CBMC. The outer horizontal and vertical lines refer to memory limit of 2GB, and the inner lines refer to timeout at 200 s.

set. The column **Solved** indicates the number of benchmarks which were solved by each tool within the time and memory limits. In total HiFrog solved 24 more benchmarks than cbmc[5]. Among 98 benchmarks for which HiFrog succeeded to return an answer within the time and memory limits, 24 benchmarks were *unsafe* and 74 benchmarks were *safe*. Interestingly, the average running time for unsafe benchmarks was longer (78 s) than the one for safe ones (48 s). This can be explained by our observation that in the unsafe cases, an iterative refinement of all the summaries was required to confirm the validity of the counter-example. However, in the safe cases, HiFrog was comparable to cbmc.

As can be seen from the column **Timeouts**, cbmc performed better than HiFrog on SV-COMP benchmarks, but it failed on almost 60% of our crafted benchmarks. As can be seen from the column **Memory outs**, HiFrog solved eight more SV-COMP benchmarks, on which cbmc immediately exceeded the memory limits. Overall, the experiments show that HiFrog is able to solve more benchmarks, and both times out and runs out of memory less often than cbmc.

Fig. 4 gives a scatter plot representing a more detailed performance comparison of HiFrog and cbmc. Each cross in the figure stands for a single benchmark with the running time of

[5] Since many of our benchmarks include non-linear arithmetic, we also tried cbmc with the experimental `--refine` option. This did not significantly change the results, and therefore we report here the results obtained with the default options of cbmc.

HiFrog on the x-axis, and the running time of cbmc on the y-axis. The crosses on the outer lines correspond to executions that exceeded the memory limit of 2GB, and the crosses on the inner lines correspond to executions that exceeded the time limit of 200 s. A large amount of crosses on the top horizontal lines lets us conclude that HiFrog is able to solve benchmarks which are challenging for cbmc. Furthermore, the solving is relatively fast in these cases.

The last three rows in Table 1 explain how our novel algorithm in HiFrog performs compared to the earlier version of HiFrog, in which summary reuse was naïve and manual with respect to successive assertions. Because this functionality was not directly available in the older HiFrog, we prepared a set of helper scripts so that the older HiFrog could process assertions one after the other with possible re-use of the summaries. As expected, EUF and LRA had a large number of unsafe results, 43 and 40 respectively. We marked such results as *unknown* since due to the abstract nature of EUF and LRA the results are possibly spurious and thus cannot be trusted. By comparison, all unsafe results returned by our new algorithm correspond to actual bugs. Verifying with BV revealed that a large number of benchmarks (56 instances) exceeded the memory limit, manifesting the cost of bit-blasting, which is avoided in our new approach whenever possible.

In conclusion, we find it encouraging that the techniques described in this paper provide such an impressive performance increase in our model checking procedure. Considering both the effectiveness and the downsides of our approach, in overall the evaluation results show a significant positive impact on the effectiveness and efficiency of verification of large-scale and multi-property benchmarks. Although we acknowledge that these initial results obtained with the 140 instances might not be enough to draw a decisive conclusion, the results do justify future efforts into extending the benchmarking, among others, to large-scale instances with multiple user-defined assertions.

8 Conclusion and Future Work

We have presented a novel SMT-based approach to incremental verification scalable to large-scale programs with multiple properties. Our key idea is to exploit both the function summaries and the overall precision of the program encoding lazily. That is, among several theories available for the encoding, we have proposed to identify the lightest one suitable for each given property. To exploit laziness, we have designed a theory interface which enables the exchange of function summaries among formulas in different theories and avoids an expensive theory combination. Thus, our proposed algorithm performs both *local refinement* and *global refinement* on demand. We were able to prove the effectiveness of our algorithm in practice, by implementing the approach on top of the HiFrog tool and carrying out an extensive experimental evaluation on the SV-COMP benchmarks. Our results show that in comparison to a state-of-the-art model checker cbmc, our tool can solve more instances within the same limits on time and memory.

Future work. In the future, we intend to study the applicability of this approach to other areas of program verification, such as upgrade checking [17], which considers a task of verification of somewhat related programs against the same property (as opposed to verification of the same program against somewhat related properties, as in the context of this paper). We also plan to apply the developed ideas in algorithms such as software verification based on IC3 [12], where correctness of unbounded programs is reduced to finding general proofs for a sequence of verification conditions that is generated on-the-fly.

Acknowledgments. This work was supported by the SNF grants 166288, and 1525936, and the Czech Science Foundation project 18-17403S.

References

[1] Aws Albarghouthi and Kenneth L. McMillan. Beautiful interpolants. In *Proc. CAV 2013*, volume 8044 of *LNCS*, pages 313–329. Springer, 2013.

[2] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Lazy abstraction with interpolants for arrays. In *Proc. LPAR 2012*, volume 7180 of *LNCS*, pages 46–61. Springer, 2012.

[3] Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even Mendoza, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina. HiFrog: SMT-based function summarization for software verification. In *Proc. TACAS 2017*, pages 207–213, 2017.

[4] Leonardo Alt, Antti E. J. Hyvärinen, and Natasha Sharygina. LRA interpolants from no man's land. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Proc. HVC 2017*, volume 10629 of *LNCS*, pages 195–210. Springer, 2017.

[5] Leonardo Alt, Antti Eero Johannes Hyvärinen, Sepideh Asadi, and Natasha Sharygina. Duality-based interpolation for quantifier-free equalities and uninterpreted functions. In Daryl Stewart and Georg Weissenbacher, editors, *Proc. FMCAD 2017*, pages 39–46. IEEE, 2017.

[6] Zaher S Andraus, Mark H Liffiton, and Karem A Sakallah. Reveal: A formal verification tool for Verilog designs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 343–352. Springer, 2008.

[7] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *Int. Conference on Software Engineering (ICSE '08)*, pages 211–220, 2008.

[8] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN Workshop (SPIN '00)*, pages 113–130, 2000.

[9] Gérard Basler, Daniel Kroening, and Georg Weissenbacher. SAT-Based Summarization for Boolean Programs. In *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings*, pages 131–148, 2007.

[10] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV 2011*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proc. TACAS 1999*, volume 1579 of *LNCS*, pages 193–207, 1999.

[12] Aaron R. Bradley. SAT-based model checking without unrolling. In *Proc. VMCAI 2011*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[13] G. Cabodi, M. Palena, and P. Pasini. Interpolation with guided refinement: Revisiting incrementality in SAT-based unbounded model checking. In *Proc. FMCAD 2014*, pages 12:43–12:50. IEEE, 2014.

[14] Gianpiero Cabodi, Marco Murciano, Sergio Nocco, and Stefano Quer. Stepping forward with interpolants in unbounded model checking. In *Proc. ICCAD 2006*, pages 772–778. ACM, 2006.

[15] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. of Symbolic Logic*, 22(3):269–285, 1957.

[16] Grigory Fedyukovich and Rastislav Bodík. Accelerating Syntax-Guided Invariant Synthesis. In *TACAS, Part I*, volume 10805 of *LNCS*, pages 251–269. Springer, 2018.

[17] Grigory Fedyukovich, Ondrej Sery, and Natasha Sharygina. Flexible SAT-based framework for incremental bounded upgrade checking. *STTT*, 19(5):517–534, 2017.

[18] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *Proc. CAV 2015*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015.

[19] Shaobo He and Zvonimir Rakamarić. Counterexample-guided bit-precision selection. In *Asian Symposium on Programming Languages and Systems*, pages 534–553. Springer, 2017.

[20] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate automizer and the search for perfect interpolants - (competition contribution). In *Proc. TACAS 2018*, pages 447–451, 2018.

[21] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In *Principles of Prog. Languages (POPL '10)*, pages 471–482, 2010.

[22] C. Hoare. Procedures and parameters: An axiomatic approach. *Symposium on Semantics of Algorithmic Languages*, pages 102–116, 1971.

[23] Antti E. J. Hyvärinen, Sepideh Asadi, Karine Even-Mendoza, Grigory Fedyukovich, Hana Chockler, and Natasha Sharygina. Theory refinement for program verification. In *Proc. SAT 2017*, volume 10491 of *LNCS*, pages 347–363. Springer, 2017.

[24] Antti E. J. Hyvärinen, Matteo Marescotti, Leonardo Alt, and Natasha Sharygina. OpenSMT2: An SMT solver for multi-core and cloud computing. In *Proc. SAT 2016*, volume 9710 of *LNCS*, pages 547–553. Springer, 2016.

[25] Radu Iosif and Xiao Xu. Abstraction refinement for emptiness checking of alternating data automata. In Dirk Beyer and Marieke Huisman, editors, *Proc. TACAS 2018*, volume 10806 of *Lecture Notes in Computer Science*, pages 93–111. Springer, 2018.

[26] Daniel Kroening and Michael Tautschnig. CBMC – C Bounded Model Checker. In Erika Ábrahám and Klaus Havelund, editors, *Proc. TACAS 2014*, pages 389–391, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[27] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV 2003*, volume 2725 of *LNCS*, pages 1–13. SV, 2003.

[28] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Proc. CAV 2006*, volume 4144 of *LNCS*, pages 123–136, 2006.

[29] Kenneth L. McMillan. Lazy annotation revisited. In *Proc. CAV 2014*, volume 8559 of *LNCS*, pages 243–259. Springer, 2014.

[30] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[31] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995.

[32] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2013.

[33] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. Interpolation-based function summaries in bounded model checking. In *Proc. HVC 2011*, volume 7261 of *LNCS*, pages 160–175. Springer, 2012.

[34] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *Proc. FMCAD 2014*, pages 1–8. IEEE, 2009.

[35] Yakir Vizel, Arie Gurfinkel, and Sharad Malik. Fast interpolating BMC. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 641–657. Springer, 2015.

[36] Yichen Xie and Alexander Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pages 139–143, 2005.

A Appendix

We define here a particular class of quantifier free theory of bit-vectors (BV) which is based on our earlier paper in [23]. In that paper the presented theory called BVP (Bit Vectors for Programs) which was an augmented version of the theory of bit-vectors. For abbreviation we use in this paper the BV notation.

In order to be applicable in our framework, BV should comply with the restriction that no overflows are allowed.

Based on SMT-LIB2 standard the signature in BV is as follows: $\Sigma^{BV} = \{+, *, bvand, bvor, bvudiv, bvurem, bvshl, bvlshr, bvnot, bvneg\}$. We consider the predicate symbols as $\mathcal{P} = \{>, <, \leq, \geq\}$. Note that for the addition and multiplication we use the same notation i.e., "+" and "*" throughout the paper to highlight the fact that the syntax are in common with our theory of interest. Therefore syntactically they can be used in the transformation rules. However, the task of interpretation of each function symbol must be delegated to the corresponding theory solver.

In the following the rules for translation from $\mathfrak{T}$ to BV are as follows:

$$\begin{gathered}
\frac{[t_1 = t_2]^{BV}}{[t_1]^{BV} = [t_2]^{BV}} \\[2ex]
\frac{[v]^{BV}}{v} \qquad v \text{ is a variable or a constant} \\[2ex]
\frac{[t_1 \bowtie t_2]^{BV}}{[t_1]^{BV} \bowtie [t_2]^{BV}} \qquad \begin{array}{l}\bowtie \text{ is a predicate symbol or binary function symbol in } BV, \\ \quad i.e., \bowtie \in \{bvand, bvor, +, *, bvudiv, bvurem, bvshl, bvlshr\}\end{array} \\[2ex]
\frac{[\triangle\, t_1]^{BV}}{\triangle\, [t_1]^{BV}} \qquad \triangle \text{ is a unary function symbol in } BV,\ i.e., \triangle \in \{bvnot, bvneg\} \\[2ex]
\frac{[f(\mathbf{t})]^{BV}}{\mathcal{M}(f(\mathbf{t}))} \text{ otherwise}
\end{gathered} \tag{8}$$

The rules for transforming from BV to theory interface $\mathfrak{T}$ are as follows:

$$\begin{gathered}
\frac{[t_1 = t_2]^{\mathfrak{T}}}{[t_1]^{\mathfrak{T}} = [t_2]^{\mathfrak{T}}} \\[2ex]
\frac{[t_1 \bowtie t_2]^{\mathfrak{T}}}{[t_1]^{\mathfrak{T}} \bowtie [t_2]^{\mathfrak{T}}} \qquad \begin{array}{l}\bowtie \text{ is a binary function symbol in } BV, \\ e.g., \bowtie \in \{bvand, bvor, +, *, bvudiv, bvurem, bvshl, bvlshr\}\end{array} \\[2ex]
\frac{[\triangle\, t_1]^{\mathfrak{T}}}{\triangle\, [t_1]^{\mathfrak{T}}} \qquad \triangle \text{ is a unary function symbol in } BV, \triangle \in \{bvnot, bvneg\} \\[2ex]
\frac{[v]^{\mathfrak{T}}}{v} \qquad v \text{ is a variable or a constant}, v \notin dom(\mathcal{M}^{-1}) \\[2ex]
\frac{[v]^{\mathfrak{T}}}{\mathcal{M}^{-1}(v)} \qquad v \in dom(\mathcal{M}^{-1})
\end{gathered} \tag{9}$$

In the following we present the translation rules from the NRA to $\mathfrak{T}$ and vice versa which are listed in (10) and (11), respectively. The rules for transforming from $\mathfrak{T}$ to NRA are as follows:

$$\frac{[t_1 = t_2]^{NRA}}{([t_1]^{NRA} \leq [t_2]^{NRA}) \wedge ([t_2]^{NRA} \leq [t_2]^{NRA})}$$

$$\frac{[v]^{NRA}}{v} \; v \text{ is a variable or a constant} \tag{10}$$

$$\frac{[t_1 \bowtie t_2]^{NRA}}{[t_1]^{NRA} \bowtie [t_2]^{NRA}} \; \bowtie \text{ is a function symbol in } NRA, e.g., \bowtie \in \{+, -, *\}$$

Likewise the LRA rules, the rules for transforming from NRA to $\mathfrak{T}$ are as follows:

$$\frac{([t_1]^{\mathfrak{T}} \leq [t_2]^{\mathfrak{T}}) \wedge ([t_2]^{\mathfrak{T}} \leq [t_1]^{\mathfrak{T}})]}{[t_1]^{\mathfrak{T}} = [t_2]^{\mathfrak{T}}}$$

$$\frac{[v]^{\mathfrak{T}}}{v} \; v \text{ is a variable or a constant} \tag{11}$$

$$\frac{[f(\mathbf{t})]^{\mathfrak{T}}}{f([\mathbf{t}]^{\mathfrak{T}})}$$