Incremental Verification by SMT-based Summary Repair

Sepideh Asadi*, Martin Blicha*†, Antti Hyvärinen*, Grigory Fedyukovich‡, Natasha Sharygina*
*Università della Svizzera italiana, Lugano, Switzerland
†Charles University, Faculty of Mathematics and Physics, Czech Republic
‡Florida State University, Tallahassee, USA
*{asadis,blicham,hyvaeria,sharygin}@usi.ch, ‡grigory@cs.fsu.edu

***Abstract*—We present UpProver, a bounded model checker designed to incrementally verify software while it is being gradually developed, refactored, or optimized. In contrast to its predecessor, a SAT-based tool eVolCheck, our tool exploits first-order theories available in SMT solvers, offering two more levels of encoding precision: linear arithmetic and uninterpreted functions, thus allowing a trade-off between precision and performance. Algorithmically UpProver is based on the reuse and repair of interpolation-based function summaries from one software version to another. UpProver leverages tree-interpolation systems in SMT to localize and speed up the checks of new versions. UpProver demonstrates an order of magnitude speedup on large-scale programs in comparison to eVolCheck and HiFrog, a non-incremental bounded model checker.**

I. Introduction

Software is always in a state of constant change. While verifying a large amount of closely related programs, a significant portion of efforts is repeated. One approach to overcome this issue is to operate incrementally by attempting to maximally reuse the results of previous computations. Furthermore, the performance and scalability of verification depends on the way software is encoded. To avoid the expensive bit-blasting during SAT-based verification, a variety of encodings offered by Satisfiability Modulo Theories (SMT) are successfully used in state-of-the-art tools. For instance, checking arithmetic properties about software might often be performed by a solver for Linear Real Arithmetic. While automatically identifying a proper level of encoding is difficult (and not a subject of this paper), tools at least should offer various encoding options to the user.

This paper presents a new tool allowing for the trade-off between efficiency and precision for the incremental analysis of pairs of software versions. *Over-approximating function summaries* are useful to enable such an analysis [1]. Summaries compactly represent all safe function behaviors, can be computed by *Craig interpolation* [2] from safety proofs of one software version, then validated on another version, and repaired if needed. An existing implementation of this idea, eVolCheck [3], uses a SAT solver and scales poorly on benchmarks that can be modeled using first-order theories. Our new Bounded Model Checking (BMC) [4] tool, called UpProver, supports several state-of-the-art SMT algorithms for interpolation [5] and allows the user to choose more efficient algorithms. In addition to the purely propositional encoding UpProver generates models with fragments of quantifier-free first-order logic, in particular in Linear Real Arithmetic (LRA) and Equality with Uninterpreted Functions (EUF). Overall, UpProver distinguishes itself by:

- Reusing the efforts invested in the verification runs of previous program versions in verification of new versions;
- Providing an ability to maintain and to repair previously computed summaries on-the-fly and to use them in the subsequent verification runs;
- Allowing for a more succinct summary representation in first-order logic as opposed to purely propositional logic;
- Leveraging the power of SMT solvers by symbolic encodings of program versions and function summaries using first-order theories (the encoding is flexible and provides an ability to adjust precision and efficiency with different levels of encoding); and
- Demonstrating a competitive performance experimentally compared to both eVolCheck and a non-incremental BMC engine while verifying gradual changes in large-scale programs.

II. Algorithmic Background

UpProver is an incremental model checker which operates on loop-free programs that are seen as a set of functions F, each $f \in F$ expressed in their Static Single Assignment form. The behavior of a program is captured by the conjunction of the SMT encodings $enc(f)$ of each $f \in F$. The program respects a safety property Q if and only if the *safety query* $\bigwedge_{f \in F} enc(f) \wedge \neg enc(Q)$ is unsatisfiable.

We use *Craig interpolation* from the proof of unsatisfiability of the safety query to construct *function summaries*, that is, relations over the input and output variables of a function that over-approximate the precise function behavior [6], [5], [7].

In UpProver, the problem of determining whether a changed program still meets the safety property, w.r.t. which the summaries were created, is reduced to the problem of validating these summaries on the changed program. To guarantee algorithmic correctness, the process requires a specialization of Craig interpolants called *tree interpolants* (see [8], [1]). The tree structure of the interpolation problem corresponds to the call tree of the program. We use approaches that guarantee the

This work was supported by Swiss National Science Foundation grant 200021_185031 and by Czech Science Foundation grant 20-07487S.

Algorithm 1: Summary validation in UPPROVER

Input: function summaries of the old version, `tree` = call-tree of the new version, Δ = set of changed functions in the new version;
Result: new version is *Safe* or *Unsafe*;

1 **while** *all* Δ *are not processed* **do**
2 choose the first f in the reverse postorder of `tree` such that $f \in \Delta$;
3 **if** *f has a summary* **then**
4 **if** *the summary is invalid* **then**
5 Remove the summary;
6 **if** *f has a parent (f is not root)* **then**
7 Add the parent to Δ to be processed;
8 **else**
9 **return** *Unsafe*, error trace;
10 **else**
11 Repair summaries from subtree of f by interpolation;
12 **return** *Safe*, set of valid and repaired summaries;

tree-interpolation property by construction [9], as opposed to, for instance, checking it on-the-fly.

Summary validation and repair, shown in Alg. 1, consists of a series of local validation checks for all changed function calls and their possibly affected callers, beginning at the deepest node. If a local validation succeeded, but for some function call in the subtree, a summary was invalidated, UPPROVER *repairs* the summary (line 11) using interpolation. Note that this local validation continues until there are no more functions to be processed, and if it succeeds, the new version is reported as *Safe*, potentially along with a set of repaired summaries that are made available for checking the next version.

It is worth noting that when the validation check propagates to the call tree root, i.e., *main* function, it corresponds to the pure BMC check where all functions are inlined. Thus in the worst case, since the programs that we check are bounded (a decidable problem), the algorithm fall backs to pure BMC.

III. Overview of UpProver

The overview of UPPROVER's architecture is shown in Fig. 1. UPPROVER implements Alg. 1 by maintaining three levels of precision—*linear real arithmetic* (LRA), *uninterpreted functions with equality* (EUF), and purely *propositional logic* (PROP)—to check the validity of summaries of program P_1 against the encodings of the function bodies of program P_2. Repaired function summaries are produced by the range of interpolation algorithms available in the underlying SMT solver. Next we describe UPPROVER's key features.

a) Efficiency / precision trade-off: A key enabler of UPPROVER's ability to adjust to user's needs in precision and efficiency is the safe over-approximation of programs with different SMT encodings. The high-level approach is to use linear or uninterpreted versions of the bit-precise program instructions whenever possible. The user selects the precision

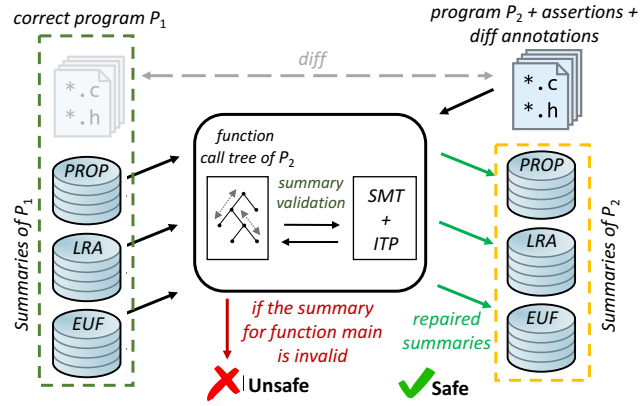

Figure 1: Overview of the UPPROVER architecture.

of the overall encoding (i.e., LRA, EUF, and PROP) which uniquely determines the precision of summaries that are available after the preceding verification of P_1.

If the user is interested in checking program properties that are likely sensitive to some bit operators in software, he/she might prefer PROP encoding. The tool then bit-blasts the program together with summaries and uses its most expensive theory (essentially, a SAT solver). However, to accelerate the process, the user might choose instead a light-weight theory, forcing the tool to pick summaries appropriately. The program statements outside of the chosen theory will be modeled nondeterministically in this case. Thus, if a bug is detected, it might be due to the theory usage, and the user is encouraged to repeat the analysis with a more precise theory (and thus, more precise summaries). Note that there is no way in general to predict the best level of encoding for each program: even if a program has seemingly bit-sensitive statements, they might be sliced out or treated nondeterministically, allowing for a successful use of a light-weight theory.

b) Summary repair: Summaries of P_1 (of the selected level of precision) are taken as input and used in the incremental analysis on demand. The tool iteratively checks if the summaries are valid for P_2 and repairs them if needed. In the best-case scenario, all summaries of P_1 are validated for P_2, copied to the persistent storage, and become available for the future analysis of P_2. When some of the summaries need repair, the tool generates new interpolants from the successful validity checks of the parent functions and stores them as the corresponding summaries. No summaries are produced when the tool returns *Unsafe*.

c) Difference annotations / validation scope: UPPROVER does not take P_1 as input, but relies on annotating the lines of code changed between P_1 and P_2. The user may choose an inexpensive syntax-level difference, or a more expensive and precise semantic-level difference that compares programs after some normalization and translation to an intermediate representation [3]. The functions that have been identified as changed are stored in Δ in Alg. 1. Note that if a function f is introduced in P_2, the caller of f is marked as changed by our difference-checker. When no summary exists for f, the

algorithm continues to check the caller. A successful validation with inlined f generates a summary for f.

d) SMT solving and interpolation engine: For answering BMC queries and the subsequent Craig interpolation, UPPROVER uses the SMT solver OPENSMT [10]. The solver generates a quantifier-free first-order interpolant as a combination of interpolants from resolution refutations [11], proofs obtained from a run of a congruence closure algorithm [12], and Farkas coefficients obtained from the Simplex algorithm in linear real arithmetic [13].

e) Implementation: UPPROVER is available as an open-source software. Each component, from difference checker to modeling to solving procedures, has been significantly optimized compared to its earlier version EVOLCHECK. As a front-end of UPPROVER, we use the infrastructure from CPROVER v5.10 to transform C program to obtain a basic unrolled BMC representation that we use as a basis for producing the final logical formula.

f) Compatibility: The summaries computed by UPPROVER are compatible with the input to HIFROG [5], another tool for incremental verification of different assertions in a *single program*. Note the difference in the use of summaries in HIFROG and UPPROVER: the former does not validate the summaries, but takes them as granted (even from the user, thus not guaranteeing the tree-interpolation property), and uses them to accelerate the verification of several new assertions.

IV. EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of UPPROVER. We demonstrate two key features of our tool: (i) the usefulness of summary reuse in verifying program revisions, and (ii) the usefulness of different levels of modeling precision, i.e., LRA, EUF, and PROP. To this end, we compare UPPROVER against our current implementation of its predecessor EVOLCHECK and a bounded model checker HIFROG.[1] Then, we compare UPPROVER with CPACHECKER [14] a tool that uses intermediate results called *abstraction precision* for caching and reusing.

Our benchmarks, containing one or more assertions, originate from different revisions of Linux kernel device drivers from [14].[2] After excluding cases where CPROVER had front-end issues, we shortlisted 1679 revision pairs (LOC on average 16K). We also included 92 pairs hand-crafted benchmarks that stress-test our implementation. All experiments were run with a 30 GB memory limit and a 600 s time limit. The complete experimental results, benchmarks, and the source code are available at http://verify.inf.usi.ch/upprover.

A. Demonstrating the effect of summary reuse in UPPROVER

In the first set of experiments, we compare UPPROVER (incremental verification) against HIFROG (non-incremental verification). The results of the experiment within the same theory encodings by LRA and EUF are displayed in Fig. 2. A large amount of points (that represent pairs of runs) on the upper triangle reveals that UPPROVER is an order of magnitude faster than the non-incremental verification in HIFROG.

Table I gives further details on 11 randomly selected pairs of benchmarks comparing the non-incremental HIFROG and incremental UPPROVER, both using the EUF encoding. Our results in LRA are very similar and therefore omitted. Each row in Table I refers to a pair (P_1, P_2) of programs. We use acronyms $|F|$ for number of functions in P_1, $|\Delta|$ for number of changed functions in P_2, *diff* for the time to construct Δ between P_1 and P_2, *itp* for the time for generating all summaries after successful bootstrapping of P_1, and *result* for reporting whether the second version was safe or unsafe. The columns *total* in HIFROG and UPPROVER show the total verification time for non-incremental and incremental verification of P_2 respectively. Even though UPPROVER's total time includes overhead such as summary repair for the subsequent runs and the difference check, in many benchmarks UPPROVER convincingly outperforms non-incremental HIFROG.

The *speedup* column demonstrates the relative speedup of UPPROVER over non-incremental verification in HIFROG. In the majority of cases, UPPROVER gains a significant speedup when reusing EUF summaries (typeset in bold). This occurs especially when the two versions have the same intermediate representation (e.g., pair 3) and the validation check is omitted. Slowdowns typically happen when both the number of changed functions and the iterative validation checks are big (e.g., pair 9), or when the verification task is relatively trivial in non-incremental verification (e.g., pairs 2 and 5). Slowdowns are demonstrated on average of 0.6x for 30% of our benchmarks in UPPROVER. On the other hand, the positive effect of summary reuse in UPPROVER was very evident, with notable speedup of 10.7x on 70% of benchmarks in LRA and EUF on average, and with an impressive max value of 109x in EUF and 104x in LRA summary reuse.

B. Demonstrating the effect of theory encoding in UPPROVER

Figure 3 illustrates the trade-off between the precision and run time of incremental verification by comparing the LRA/EUF-based encodings in UPPROVER against the PROP-based encoding in EVOLCHECK. Each point corresponds to an incremental verification run on P_2. Almost universally, whenever run time exceeds one second, it is an order of magnitude faster to verify with LRA and EUF than

Table I: UPPROVER using EUF summary vs. non-incremental HIFROG.

		boot	**HIFROG**		**UpProver**				
pair	$\|F\|$	$itp(s)$	$total(s)$	$diff(s)$	$\|\Delta\|$	$*valid$	$total(s)$	*speedup*	*result*
1	2124	0.5	36.6	0.2	80	92	8.9	**4.1**	Safe
2	25	0.1	0.8	0.1	1	2	1.8	0.4	Unsafe
3	2291	1.3	41.8	0.2	0	0	0.5	**92.9**	Safe
4	2148	0.6	41.4	0.2	4	4	0.8	**55.2**	Safe
5	544	0.1	2.4	0.1	95	105	4.7	0.5	Unsafe
6	4350	0.7	32.1	0.5	415	552	58.1	0.6	Safe
7	665	0.1	2.5	0.1	1	1	0.2	**10.8**	Safe
8	357	0.1	3.8	0.1	10	13	0.4	**9.9**	Safe
9	5417	0.6	43.2	0.5	750	1201	101.1	0.4	Safe
10	2121	0.5	37.6	0.2	4	4	0.8	**49.5**	Safe
11	31246	3.2	83.2	12.1	30	41	78.2	**1.1**	Safe

[1]The tools share the same front-end for parsing C programs, thus the comparison is not affected by unrelated implementation differences.

[2]https://www.sosy-lab.org/research/RegressionVerification/.

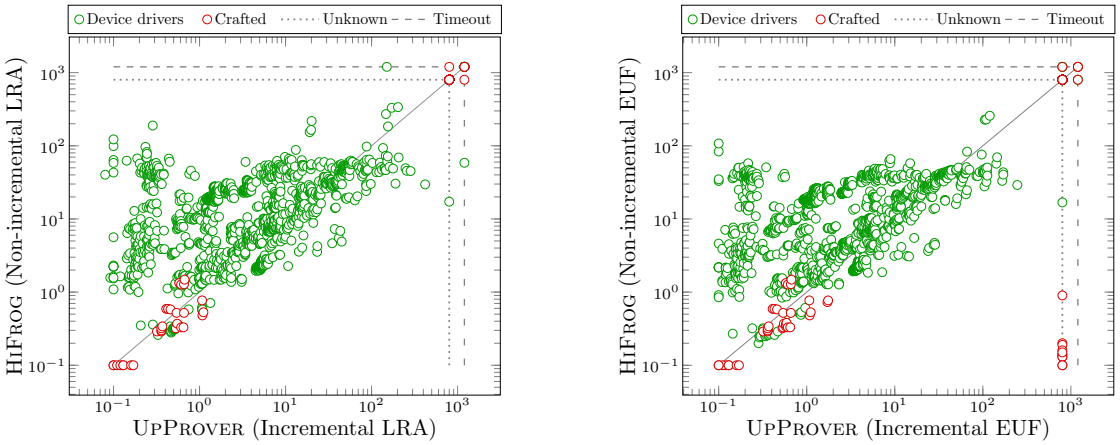

Figure 2: Demonstrating the impact of summary reuse by comparing verification time of UpProver versus HiFrog on LRA (*left*) and EUF (*right*).

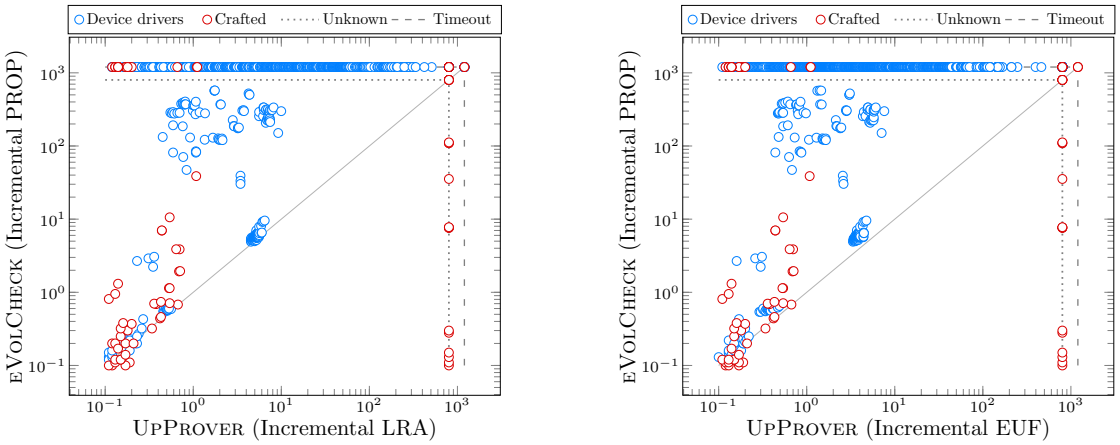

Figure 3: Demonstrating the impact of theory encoding by comparing timings of LRA/EUF encodings in UpProver vs. PROP encoding in eVolCheck.

Table II: A comparison of different encodings in UpProver on device drivers (light gray) and crafted benchmarks (dark gray).

	PROP		**LRA**		**EUF**		**Regret PROP**	**Regret LRA**	**Regret EUF**
results	P_1	P_2	P_1	P_2	P_1	P_2	P_1+P_2	P_1+P_2	P_1+P_2
Safe	353	353	1591	1589	1591	1590	0+0	0+0	0+0
Unsafe	0	0	78*	1*	85*	1*	n/a	n/a	n/a
TO	1326	2	10	1	3	0	n/a	n/a	n/a
Total	1679								
Safe	57	38	73	57	35	28	2+2	16+17	4+6
Unsafe	6	12	16*	12*	53*	6*	n/a	n/a	n/a
TO	27	3	3	0	4	0	n/a	n/a	n/a
MO	2	0	0	0	0	0	n/a	n/a	n/a
Total	92								

with PROP. In addition, a large number of benchmarks on the top horizontal lines suggests that it is possible to solve many more instances with LRA/EUF-based encoding than with PROP-based encoding. However, the loss of precision is seen on the benchmarks on the vertical line labeled Unknown, indicating if the incremental result using LRA/EUF is unsafe, the result might be spurious because of abstraction.

More statistics are shown in Table II. For each encoding within the time and memory limits, the benchmarks are reported as *Safe* or *Unsafe*. The unsafe results might be spurious on theory encodings (indicated by an asterisk). We use acronyms *TO* for time out, *MO* for memory out, and P_1, P_2 for two versions of a program. We notice that PROP times out in 76% of the benchmarks, while LRA and EUF time out for less than 1%. The last three columns (the first number refers to P_1 and the second to P_2) indicate how many benchmarks can be solved exclusively in a single encoding. This can be interpreted as the *regret* of not including a solver in an imaginary portfolio.

The results for crafted benchmarks show that the theories are complementary, with LRA having the biggest regret. This can be contrasted to the plot in Fig. 4 showing that LRA encoding has a constant 30% time overhead compared to EUF due to the more expensive decision procedure.

Our extensive experimentation reveals that LRA and EUF encodings are crucial for scalability. At the same time, there is a small number of benchmarks that require PROP. While it is unsurprising that bit-blasted models are more expensive to check than the EUF and LRA models, we find it surprising that the light-weight encodings work so often. In effect, the encodings complement each other, and the results suggest an approach where the user gradually tries different precisions until one is found that suits the benchmarks at hand.

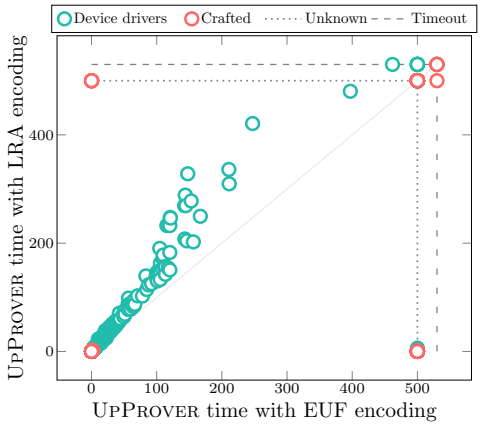

Figure 4: LRA vs. EUF in UpProver.

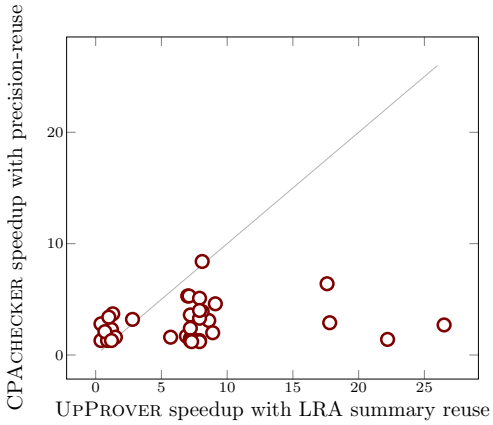

Figure 5: Speedup in UpProver vs. CPAchecker.

C. Comparison of UpProver and CPAchecker

Finally, we compare UpProver with a widely-used tool CPAchecker that is able to perform incremental verification by reusing abstraction precisions. It is an orthogonal technique to ours, i.e., it is an unbounded verifier and aims at finding loop invariants. Thus, comparing running times does not make sense since running times in UpProver crucially depend on the chosen bound.[3] Instead, we concentrate on comparing the speedups obtained with the two methods since the change of a bound affects a speedup less.

Here we report the results only on device driver instances which both tools could handle. Among the 250 device drivers categories reported in https://www.sosy-lab.org/research/cpa-reuse/predicate.html, we matched 34 categories which are suitable for UpProver.[4] These categories contain in total 903 verification tasks.

Fig. 5 depicts the comparison of speedup in UpProver and CPAchecker. A large amount of points on the lower triangle lets us conclude that summary reuse in UpProver achieves superior speedup than the precision reuse in CPAchecker. The average speedup in UpProver was 7.3 with standard-deviation of 6 and in CPAchecker the average speedup was 2.9 with standard-deviation of 1.7. There were 4 slowdowns in UpProver whereas CPAchecker did not report any slowdowns on these 34 categories. The detailed results are available at http://verify.inf.usi.ch/upprover/experimentation.

[3]For instance, the average running times in CPAchecker is 285.3 seconds and in UpProver with LRA is 13.4 seconds for chosen bound 5. For other bounds UpProver would have different average running times.

[4]The reported version of UpProver is constrained by its dependency on the CProver framework which impedes its frontend from processing some benchmarks.

V. Related Work

The trend towards constructing efficient tools for incremental formal verification exists since last two decades [15]. We identify here two main approaches to incremental verification of different revisions of a program:

a) Differential program reasoning: Reasoning over multiple programs (to, e.g., prove program equivalence) is usually performed by creating a so-called *product program* [16], [17], [18], [19], [20], [21] and analyzing this product program using the general-purpose tools. These approaches, however, do not usually consider properties about isolated programs. A modular approach that works by simultaneously traversing the call trees of both programs is proposed in [16], but it does not use function summaries. A probabilistic framework has been recently proposed in [22], but it is applicable to differential bug finding, rather than to proving the absence of bugs.

b) Incremental Verification: A number of approaches accelerate verification by reusing previous efforts. Program changes are extensively used in incremental modal μ-calculus [23], solving of Constrained Horn Clauses [24], [25], predicate abstraction [14], [26], automata-based approaches [27], reusing the results from constraint solving [28], and state-space graph for checking temporal safety properties [29]. However these groups of techniques are orthogonal to our approach as we store and reuse the interpolation-based function summaries in the context of BMC for verifying revisions of programs. In addition, our tool outputs a *certificate of correctness* in the form of a function summary that can be used as a function specification.

VI. Conclusion and Future Work

We presented UpProver, an SMT-based incremental BMC tool for different revisions of a program. Its key innovation is in several SMT-level encodings and the corresponding SMT-level summarization algorithms that allow the user to adjust the precision or efficiency of verification. UpProver enables LRA and EUF theories (and in the future, more) thus allowing a trade-off between precision and performance. Furthermore, our approach not only extracts function summaries but provides a capability of repairing them on-the-fly and reusing them in the subsequent verification runs. Our experimentation reveals that UpProver is more efficient than its predecessor and the two orthogonal approaches: non-incremental bounded model checker [5] and precision reuse [14].

In future we extend the tool to handle summaries from different theories simultaneously in the style of [7] and [30], possibly by allowing checks for the tree-interpolation property on-the-fly.

References

[1] O. Sery, G. Fedyukovich, and N. Sharygina, "Incremental upgrade checking by means of interpolation-based function summaries," in *Proc. FMCAD 2012*. IEEE, 2012, pp. 114–121.

[2] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," *J. Symb. Log.*, vol. 22, no. 3, pp. 269–285, 1957.

[3] G. Fedyukovich, O. Sery, and N. Sharygina, "eVolCheck: Incremental upgrade checker for C," in *Proc. TACAS 2013*, ser. LNCS, vol. 7795. Springer, 2013, pp. 292–307.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. TACAS 2003*, ser. LNCS, vol. 1579. Springer, 1999, pp. 193–207.

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

[6] O. Sery, G. Fedyukovich, and N. Sharygina, "Interpolation-based function summaries in bounded model checking," in *Proc. HVC 2011*, ser. LNCS, vol. 7261. Springer, 2011, pp. 160–175.

[7] S. Asadi, M. Blicha, G. Fedyukovich, A. E. J. Hyvärinen, K. Even-Mendoza, N. Sharygina, and H. Chockler, "Function summarization modulo theories," in *Proc. LPAR-22*, ser. EPiC Series in Computing, vol. 57. EasyChair, 2018, pp. 56–75.

[8] S. Asadi, M. Blicha, A. E. J. Hyvärinen, G. Fedyukovich, and N. Sharygina, "Farkas-based tree interpolation," in *Proc. SAS 2020 (to appear)*, ser. LNCS. Springer.

[9] J. Christ and J. Hoenicke, "Proof tree preserving tree interpolation," *J. Autom. Reasoning*, vol. 57, no. 1, pp. 67–95, 2016.

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

[11] L. Alt, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina, "A proof-sensitive approach for small propositional interpolants," in *Proc. VSTTE 2015*, ser. LNCS, vol. 9593. Springer, 2016, pp. 1–18.

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

[13] M. Blicha, A. E. J. Hyvärinen, J. Kofron, and N. Sharygina, "Decomposing Farkas interpolants," in *Proc. TACAS 2019*, ser. LNCS, vol. 11427. Springer, 2019, pp. 3–20.

[14] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler, "Precision reuse for efficient regression verification," in *Proc. ESEC/FSE 2013*. ACM, 2013, pp. 389–399.

[15] R. H. Hardin, R. P. Kurshan, K. L. McMillan, J. A. Reeds, and N. J. A. Sloane, "Efficient regression verification," in *Proc. WODES'96*. IEEE, 1996, pp. 147–150.

[16] B. Godlin and O. Strichman, "Regression verification," in *Proc. DAC 2009*. ACM, 2009, pp. 466–471.

[17] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *Proc. FM 2011*, ser. LNCS, vol. 6664. Springer, 2011, pp. 200–214.

[18] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel, "Differential assertion checking," in *Proc. FSE 2013*. ACM, 2013, pp. 345–355.

[19] L. Pick, G. Fedyukovich, and A. Gupta, "Exploiting synchrony and symmetry in relational verification," in *Proc. CAV 2018, Part I*, ser. LNCS, vol. 10981. Springer, 2018, pp. 164–182.

[20] R. Shemer, A. Gurfinkel, S. Shoham, and Y. Vizel, "Property directed self composition," in *CAV 2019, Part I*, vol. 11561. Springer, 2019, pp. 161–179.

[21] D. Mordvinov and G. Fedyukovich, "Property directed inference of relational invariants," in *Proc. FMCAD 2019*. IEEE, 2019, pp. 152–160.

[22] K. Heo, M. Raghothaman, X. Si, and M. Naik, "Continuously reasoning about programs using differential Bayesian inference," in *Proc. PLDI 2019*. ACM, 2019, pp. 561–575.

[23] O. Sokolsky and S. A. Smolka, "Incremental model checking in the modal mu-calculus," in *Proc. CAV 1994*, ser. LNCS, vol. 818. Springer, 1994, pp. 351–363.

[24] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Incremental verification of compiler optimizations," in *Proc. NFM 2014*, ser. LNCS, vol. 8430. Springer, 2014, pp. 300–306.

[25] ——, "Property directed equivalence via abstract simulation," in *Proc. CAV 2016*, vol. 9780, Part II. Springer, 2016, pp. 433–453.

[26] F. He, Q. Yu, and L. Cai, "When regression verification meets CEGAR," *CoRR*, vol. abs/1806.04829, 2018. [Online]. Available: http://arxiv.org/abs/1806.04829

[27] B. Rothenberg, D. Dietsch, and M. Heizmann, "Incremental verification using trace abstraction," in *Proc. SAS 2018*, ser. LNCS, vol. 11002. Springer, 2018, pp. 364–382.

[28] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: reducing, reusing and recycling constraints in program analysis," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12,*. ACM, 2012, p. 58.

[29] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido, "Extreme model checking," in *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, 2003, pp. 332–358.

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