HiFrog: SMT-based Function Summarization for Software Verification

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

[1] Università della Svizzera italiana, Switzerland
[2] King's College London, UK
[3] University of Washington, USA

Abstract. Function summarization can be used as a means of incremental verification based on the structure of the program. HiFrog is a fully featured function-summarization-based model checker that uses SMT as the modeling and summarization language. The tool supports three encoding precisions through SMT: uninterpreted functions, linear real arithmetics, and propositional logic. In addition the tool allows optimized traversal of reachability properties, counter-example-guided summary refinement, summary compression, and user-provided summaries. We describe the use of the tool through the description of its architecture and a rich set of features. The description is complemented by an experimental evaluation on the practical impact the different SMT precisions have on model-checking.

1 Introduction

Incremental verification addresses the unique opportunities and challenges that arise when a verification task can be performed in an incremental way, as a sequence of smaller closely related tasks. We present an implementation of the incremental verification of software with assertions that uses the insights obtained from a successful verification of earlier assertions. As a fundamental building block in storing the insights we use function summaries known to provide speedup through localizing and modularizing verification [9,13].

In this paper we describe the HiFrog verification tool that uses Craig interpolation [6] in the context of Bounded Model Checking (BMC) [4] for constructing function summaries. The novelty of the tool is in the unique way it combines function summaries with the expressiveness of satisfiability modulo theories (SMT). The system currently supports verification based on the quantifier-free theories of linear real arithmetics (QF_LRA) and uninterpreted functions (QF_UF), in addition to propositional logic (QF_BOOL). Compared to our earlier tool FunFrog [9] constructing function summaries in the propositional logic, the SMT summaries are smaller and more efficient in verification. They are also often significantly more human-readable, enabling their easier reuse, as well as injection of summaries provided directly by the user. In addition, the tool offers a rich set of features such as verification of recursive programs, different ways of optimizing the summaries with respect to both size and

strength, efficient heuristics for removing redundant safety properties, and easy-to-understand witnesses of property violations that can be directly mapped to bugs in the source code.

The paper provides an architectural description of the tool, an introduction to its use, and experimental evidence of its performance. The tool together with a comprehensive demo is available at `http://verify.inf.usi.ch/hifrog`.

Related work. Incremental verification is a subject of extensive research in different domains, such as hardware verification, deductive verification, and model checking. Due to the lack of space, here we provide recent related work close to our problem domain. The CPAchecker tool is able to migrate predicates across program versions [3]. Deductive verification tools such as Viper and Dafny offer modular verification [12] and cashing the intermediate verification results [10] respectively. In the context of software symbolic model checking, the closest body of work is CBMC – a bounded model-checker for C that to a limited extent exploits incremental capabilities of a SAT solver[4], but does not use or output any reusable information like function summaries. The ESBMC tool, like HiFrog, also shares the CProver infrastructure, is based on an SMT solver, but to the best of our knowledge, does not support incremental verification [5].

2 Tool Overview

HiFrog consists of two main components *SMT encoder* and *interpolating SMT solver*; and the function *summaries* (see Fig. 1). The components are configured by initializing the theory and the interpolation algorithms. The assertions are processed sequentially using the function summaries, when possible. The results of a successful verification of an assertion are stored as interpolated function summaries, and failed verifications trigger a refinement phase or the printing of an error trace. In this section we describe the features of the tool in more detail.

Preprocessing. The source code is parsed and transformed into an intermediate *goto-program* using the goto-cc symbolic compiler. The loops are unwound to the pre-determined number of iterations. HiFrog identifies the set of assertions from the source code, reads the user-defined function summaries (if any) in the smtlib2-format, and makes them available for the subsequent analysis.

SMT encoding and Function Summarization. For a given assertion, the goto-program is *symbolically executed* function-per-function resulting in the "modular" Static Single Assignment (SSA) form of the unwound program, i.e., a form where each function has its own isolated SSA-representation. To reduce the size of the SSA form, HiFrog performs *slicing* that keeps only the variables in the SSA form that are syntactically dependent on the variables in the assertion.

When the SSA form is pruned, HiFrog creates the SMT formula in the pre-determined logic (QF_BOOL, QF_UF or QF_LRA). The modularity of the SSA

[4] `http://www.cprover.org`

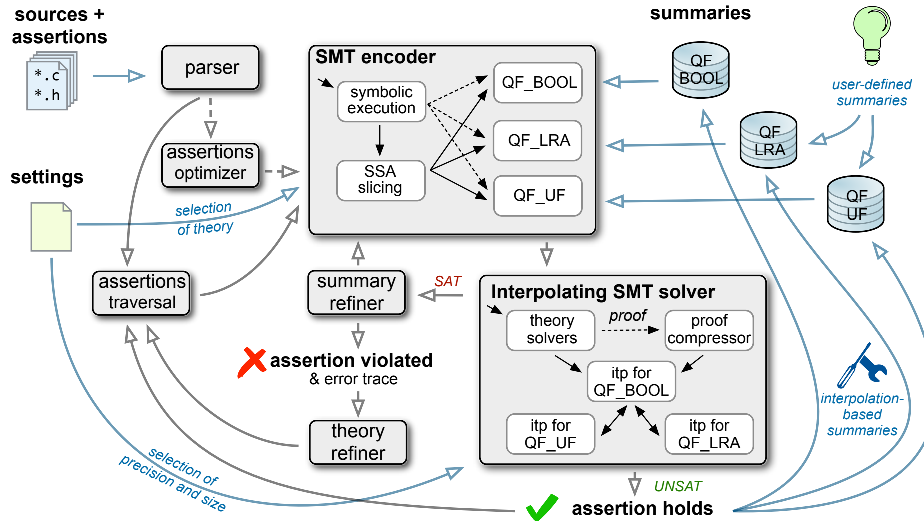

Fig. 1. HiFrog overview. Grey and black arrows connect different modules of the tool (dashed - optional). Blue arrows represent the flow of the input/output data.

form comes in handy when the function summaries of the chosen logic (either user-defined, interpolation-based, or treated nondeterministically) are available. If this is the case, the call to a function with the available summary is replaced by the summary. The final SMT formula is pushed to an SMT solver to decide its satisfiability.

Due to over-approximating nature of function summaries, the program encoded with the summaries may contain spurious errors. The *summary refiner* identifies and marks summaries directly involved in the model generated by the SMT solver. This way, HiFrog gets back to the encoding stage and replaces the involved summary by the precise encoding of the function (for recursive functions, it unrolls the function body one more time and treats recursive calls nondeterministically). For an unsatisfiable SMT formula, HiFrog extracts function summaries using interpolation. The extracted summaries are serialized in a persistent storage so that they are available for other HiFrog runs.

Theories. HiFrog supports three different quantifier-free theories in which the program can be modelled: bit-precise QF_BOOL, QF_UF and QF_LRA. The use of theories beyond QF_BOOL allows the system to scale to larger problems since encoding in particular the arithmetic operations using bit-precision can be very expensive. As the precise arithmetics often do not play a role in the correctness of the program, substituting them with linear arithmetics, uninterpreted functions, or even nondeterministic behavior might result in a significant reduction in model-checking time (see Sec. 3). If a property is proved using one of the light-weight theories QF_UF and QF_LRA, the proof holds also for the ex-

act BMC encoding of the program. However, the loss of precision can sometimes produce spurious counterexamples due to the over-approximating encoding. The light-weight theories therefore need to be refined (i.e., using *theory refiner*) to QF_BOOL if the provided counter-example does not correspond to a concrete counterexample.

Obtaining summaries by interpolation. HiFrog relies on different interpolation frameworks for the different theories it supports. As a result the generation of propositional, QF_UF and QF_LRA interpolants can be controlled with respect to strength and size by specifying an interpolation algorithm for a theory. For propositional logic we provide the *Labeled Interpolation Systems* [7] including the *Proof-Sensitive* interpolation algorithms [1]. Interpolation for QF_UF is implemented with *duality-based interpolation* [2], and a similar extension is applied to the interpolation algorithm for QF_LRA based on [11]. HiFrog also provides a range of techniques to reduce the size of the generated interpolants through removing redundancies in propositional proofs [13]: (i) the RecyclePivotsWith-Intersection (RPI) algorithm, (ii) the LowerUnits (LU) algorithm, (iii) structural hashing (SH), (iv) and a set of local rewriting rules.

Assertion Optimizer. In addition to incremental verification of a set of assertions, HiFrog supports the basic functionality of classical model checkers to verify all assertions at once. For the cases when the set of assertions is too large, it can be optimized by constructing an *assertion implication relation* and exploiting it to remove redundant assertions [8]. In a nutshell, the assertion optimizer considers pairs of spatially close assertions a_i and a_j and uses the SMT solver to check if a_i conjoined with the code between a_i and a_j implies a_j (if there is any other assertion between a_i and a_j then it is treated as assumption). If the check succeeds then a_j is proven redundant and its verification can be safely skipped.

3 HiFrog Usage

We provide a Linux binary of HiFrog which runs from the console. The binary receives as input from the user a C-program with assertions to be verified, a set of parameters and the function summaries (either interpolation-based or user-defined) in the SMT-LIB Standard v. 2.0.

HiFrog exploits the CProver framework and inherits some of its options (e.g., `--unwind` for the loop unrolling, `--show-claims` and `--claim` for managing the assertions checks); the ability for the user to declare and to use a `nondet_TYPE()` function of a specific numerical type (e.g., int, long, double, unsigned, in QF_LRA only) or add a `__CPROVER_assume()` statement to limit the domain to a specific range of values.

HiFrog uses QF_LRA by default but can be switched to QF_UF via the `--logic` option. [5] HiFrog uses a variety of interpolation and proof compression algorithms to control the the precision (with `--itp-uf-algorithm` option

[5] Currently the support for QF_BOOL needs to be specified at compile time.

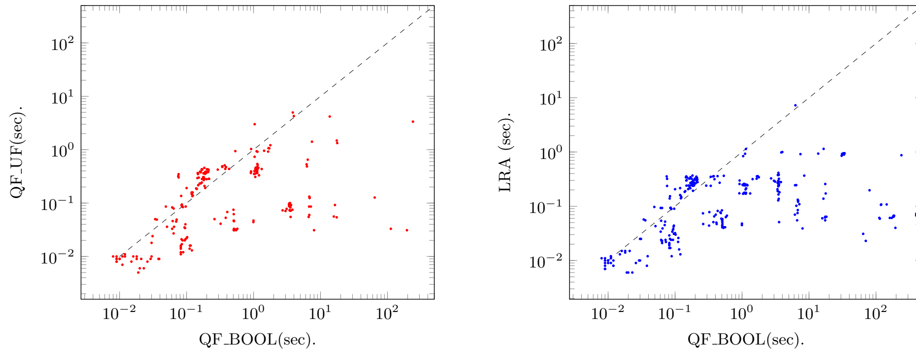

Fig. 2. Running time by QF_BOOL against QF_UF and QF_LRA.

for QF_UF, `--itp-lra-algorithm` option for QF_LRA, and `--itp-algorithm` option for propositional interpolation) and the size (with `--reduce-proof`) of summaries. The summary storage is controlled using the `--save-summaries` and `--load-summaries` options. In between verification runs, the summaries contained in the corresponding files for QF_UF and QF_LRA might be edited manually. Note that due to the SMT encoding constraints HiFrog does not allow interchanging summaries between the theories. Finally, HiFrog supports the identification and reporting of redundant assertions with `--claims-opt`, a useful feature for some automatically generated assertions [8].

In the end of each verification run, HiFrog either reports `VERIFICATION SUCCESSFUL` or `VERIFICATION FAILED` accompanied by an error trace. An error trace presents a sequence of steps with a direct reference to the code and the values of variables in these steps. In most cases when QF_UF and QF_LRA introduce a spurious error, HiFrog outputs a warning, and thus the user is advised to use HiFrog with a more precise theory. HiFrog also reports the statistics on the running time and the number of the summary-refinements performed.

Experimental Results. We evaluated HiFrog on a large set of C programs coming from both academic and industrial sources such as SV-COMP. All benchmarks contained multiple assertions to be checked. To demonstrate the advantages of the SMT-based summarization, here we provide data for analysis of benchmarks containing 1086 assertions from which 474 were proven to hold using QF_BOOL(meaning that those properties satisfy the system specifications). Even despite the over-approximating nature of QF_UF and QF_LRA, our experiments witnessed a large amount of properties which were also proven to be correct by employing the light-weight theories of HiFrog (namely, 50.65% and 69.2% of validated properties out of 474 for QF_UF and QF_LRA respectively).

Furthermore, those experiments revealed that model checking using the QF_UF and QF_LRA-based summarization was extremely efficient. Fig. 2 presents two

logarithmic plots for comparison of running times[6] of HiFrog with QF_BOOL to respectively QF_UF and QF_LRA. Each point represents a pair of verification runs of a holding assertion with the two corresponding theories using the interpolation-based summaries. Note that for most of the assertions, the verification with QF_UF and QF_LRA is in order of magnitude faster than the verification with QF_BOOL.

References

1. Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: A Proof-Sensitive Approach for Small Propositional Interpolants. In: VSTTE. LNCS, vol. 9593, pp. 1–18. Springer (2015)
2. Alt, L., Hyvärinen, A.E.J., Sharygina, N.: Duality-based interpolation for quantifier-free equalities and uninterpreted functions (2016), submitted manuscript. A version is made available for reviewers at `http://www.inf.usi.ch/postdoc/hyvarinen/euf-interpolation.pdf`
3. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: ESEC/FSE. pp. 389–399. ACM (2013)
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Tools and Algorithms for the Analysis and Construction of Systems. LNCS, vol. 1579. Springer-Verlag (1999)
5. Cordeiro, L.C., de Lima Filho, E.B.: Smt-based context-bounded model checking for embedded systems: Challenges and future trends. ACM SIGSOFT Software Engineering Notes 41(3), 1–6 (2016)
6. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. In: J. of Symbolic Logic. pp. 269–285 (1957)
7. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: Proc. VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer (2010)
8. Fedyukovich, G., Callia D'Iddio, A., Hyvärinen, A.E.J., Sharygina, N.: Symbolic Detection of Assertion Dependencies for Bounded Model Checking. In: FASE. LNCS, vol. 9033, pp. 186–201. Springer (2015)
9. Fedyukovich, G., Sery, O., Sharygina, N.: eVolCheck: Incremental upgrade checker for C. In: Proc. TACAS '13. LNCS, vol. 7795, pp. 292–307. Springer (2013)
10. Leino, K.R.M., Wüstholz, V.: Fine-grained caching of verification results. In: CAV. LNCS, vol. 9206, pp. 380–397 (2015)
11. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)
12. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: VMCAI. LNCS, vol. 9583, pp. 41–62. Springer (2016)
13. Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: PeRIPLO: A framework for producing effective interpolants in SAT-based software verification. In: LPAR. LNCS, vol. 8312, pp. 683–693. Springer (2013)

[6] The timing results were obtained on an Ubuntu 14.04.1 LTS server with Intel(R) Xeon(R) $E5620$ $CPU@2.40GHz$, 16 cores processor and 16GB RAM. We prepared a pre-compiled Linux-binary available at the Virtual Machine at `http://verify.inf.usi.ch/hifrog/binary`; our benchmarks set is available at `http://verify.inf.usi.ch/hifrog/bench` and can facilitate the property verification for other researchers.

A Tool Demo

A.1 Overview

In this section we demonstrate the basics of using HiFrog on some illustrative examples. Throughout the example, note that the HiFrog environment should be prepared for analysis by cleaning the repository with

```
$ rm __summaries
```

before performing steps that do not use previous summaries. This is important especially when verifying a new benchmark, as the summaries of previously verified benchmarks must be removed.

HiFrog uses the CProver framework. All loops and a recursive function calls in the program should be unrolled using the command line `--unwind <N>`, where `<N>` is the number of loop iterations and the recursive depth. Any non-defined function is used to draw random values of an Integer, and any declared-only function is used to draw random values of a specific type (commonly denoted as `nondet_Int()`, `nondet_Long()`, etc.). The assumptions on the code for the verification process are given using `__CPROVER_assume()` notation, and thus, we can limit the values of non-deterministically chosen variables to a specific range or interval.

A.2 Basic Functionality of HiFrog

In this section, we explain how HiFrog constructs a function summary for an assertion (also referred to as to claim) and reuses it for another claim. The first example, shown in Fig. 3, consists of a function that randomly gets 1000 integers and returns their sum.

Running your claim. In this example, we add assertions to the C-program that HiFrog verifies. The C-program with three assertions is shown in Fig. 3.

Run HiFrog to check the the first and the third assertion as follows:

```
$ rm __summaries
$ ./hifrog --logic qflra ex1_lra.c --claim 1
$ ./hifrog --logic qflra ex1_lra.c --claim 3
```

Note that `__summaries` should not be removed after running the first claim.

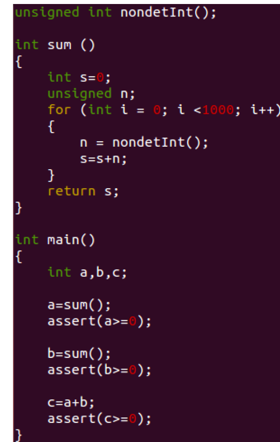

Fig. 3. `ex1_lra.c`

Fig. 4 shows the relevant parts of the output from these two checks. On both figures, HiFrog indicates that the program is safe, reporting `VERIFICATION SUCCESSFUL`. Note that despite function `nondetInt` is declared-only, HiFrog is able to automatically identify its return type (`unsigned int`) and exploit it for verification. We can also see the solver time and total time for checking these claims. The run time for the third claim is significantly lower than for the first claim due to summaries usage.

```
SOLVER TIME: 4.595
UNSAT - it holds!
ASSERTION IS TRUE
Start generating interpolants...
INTERPOLATION TIME: 0.039
ASSERTION(S) HOLD(S)


VERIFICATION SUCCESSFUL
Initial unwinding bound: 0
Total number of steps: 1
Unwinding depth: 1 (1)
TOTAL TIME FOR CHECKING THIS CLAIM: 5.245
#X: Done.
```

```
SOLVER TIME: 0.002
UNSAT - it holds!
ASSERTION IS TRUE
Start generating interpolants...
INTERPOLATION TIME: 0
FUNCTION SUMMARIES (for 3 calls) WERE SUBSTITUTED SUCCESSFULLY.


VERIFICATION SUCCESSFUL
Initial unwinding bound: 0
Total number of steps: 1
Unwinding depth: 1 (1)
TOTAL TIME FOR CHECKING THIS CLAIM: 0.015
#X: Done.
```

Fig. 4. Results of running HiFrog on `ex1_lra.c` (claim 1 and claim 3)

What actually happened. When checking the second or third assertion HiFrog reuses the previous verification results for verifying the new claims.

After the successful verification of the first assertion, HiFrog starts to generate the summaries which are stored for the subsequent verifications in the file `__summaries`. Since we specified the logic `qflra`, HiFrog generates the summaries in QF_LRA. We encourage the user to view the `__summaries` file. When checking the third assertion the generated summaries were substituted instead of encoding the function into the solver, and the speed-up we observe results from this. Since the verification of the third claim was successful, HiFrog updates the `__summaries` file with new function summaries.

A.3 Advanced Functionality of HiFrog

HiFrog offers a number of interesting features that set it apart from many other model checkers. In this section, we discuss the use of user-provided summaries, specification of different theories for modeling, automatically removal of some redundant assertions, and options for controlling the summary generation through interpolation.

User-Provided Summaries. HiFrog provides several approaches that can be used if the logic used for modeling is not sufficiently expressive. For example, this might happen when a user has non-linear arithmetics in the program. The LRA implementation of HiFrog will attempt to prove these properties by substituting the results of the unsupported operations with completely nondeterministic behaviour to maintain soundness at the expense of providing spurious counter-examples, but sometimes this is not sufficient.

```
#include <math.h>

double nondet();

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

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

Fig. 5. `sin_cos.c`

One way around this problem is to use the feature that allows the users to insert their own summaries for verification. These are provided in the SMT-LIB v2 format. To shed light on this issue, suppose that there is a C-code, shown in Fig. 5 which uses trigonometric functions and calculates $Sin^2(x) + Cos^2(x)$.

Since $Sin^2(x) + Cos^2(x) = 1$ for any x, as it is a known trigonometric identity, we can utilize this knowledge and substitute the formula with a summary

stating exactly this identity. To help the user getting started with the summary construction we provide a command for constructing a template.

```
./hifrog --logic qflra --list-templates sin_cos.c
```

```
(define-fun |c::nonlin#2| ( (|c::nonlin::x!0| Real) (|hifrog::?fun_start| Bool) (|hifrog::?fun_end| Bool)
(|c::nonlin::?retval| Real) ) Bool (let ((?def0 true))

?def0
))
```

Fig. 6. Example template for `nonlin`

This command dumps a list of templates for all functions used in the program into the `__summaries` file. Fig. 6 shows one of such the automatically generated templates that contains the `define-fun` statement, followed by the function name (`nonlin`), the set of parameters, and the body of the function which is empty (it is indicated by `true`). In Fig. 7 we have edited the template file to a new function summary which states that the return value of the function is 1. The user can now link the summary of functions to the code `sin_cos.c` as follows.

```
(define-fun |c::nonlin#0| (
  (|c::nonlin::x!0| Real)
  (|hifrog::?fun_start| Bool)
  (|hifrog::?fun_end| Bool)
  (|c::nonlin::?retval| Real) )  Bool
  (= 1 |c::nonlin::?retval|)
)
```

Fig. 7. The user-provided function summary stating that the return value of the function `nonlin` is one.

```
$./hifrog sin_cos.c --load-summaries __summaries_sin_cos
```

Intuitively, the use of user-defined summaries is to some extent similar to the use of user-defined assumptions. However, while assumptions just add additional constraints to the SMT formula and do not affect encoding of the original code, our summaries are used to replace the code completely, thus (in program `sin_cos.c`) avoiding deal with complex nonlinear constraints.

Use of Uninterpreted Functions. Fig. 8 shows a function that multiplies two variables. Because of the non-linearity, LRA is not able to verify this program. Even though non-linear SMT solvers can verify such operations, it is usually costly and not supported by many solvers. The program could also be encoded using propositional logic, but due to the complicity of multiplication encoding this turns out to be expensive.

```
int prod(int a, int b)
{
    int p = 1;
    int i;
    for (i = 0; i < 3; i++)
    {
        p = p * (a + b);
    }
    return p;
}

int main(int argc, char** argv)
{
    int a = nondet();
    int b = a;
    int c = a;
    int d = a;
    int q = prod(a, b);
    int p = prod(c, d);
    printf("p: %d; q: %d\n", p, q);
    assert(p == q);
}
```

Fig. 8. `ex1_uf.c`

However, for this particular example, the correctness of the program does not depend on the exact interpretation of the

multiplication. In fact, it suffices to assume that a function returns the same value when invoked with the same parameters. In the following we verify the program specifying the logic QF_UF.

```
$ ./hifrog --claim 1 --logic qfuf ex1_uf.c
```

Use of Propositional Logic. In some cases it is necessary to resort to the bit-precise modelling of the software through propositional logic despite the increased complexity this implies. This is supported in HiFrog currently through a separately built binary. For this particular example, the propositional logic check is done by running

```
$ ./hifrog-prop --claim 1 ex1_uf.c
```

Simplifying the Life of Users. Here we outline various optimizations and unique features of HiFrog that can be useful in different stages of verification.

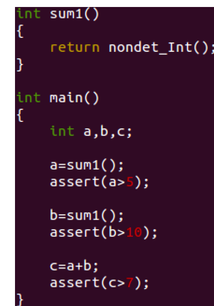

Fig. 9. ex2_lra.c

Removing redundant assertions. Fig. 9 shows a program calling a nondetermistic function sum1. Thus, the three assertions in the program are violated. So the user can get a counter-examples for each of them. Additionally, the user might be interested in running a dependency analysis to reveal other useful information about the assertions. In particular, HiFrog has an option --claims-opt <steps> which identifies and reports the redundant assertions using the threshold <steps> as the maximum number of SSA steps between the assertions including the SSA steps at the functions calls (if any) between the assertions:

```
$ ./hifrog --claims-opt 20 ex2_lra.c
```

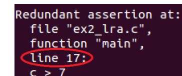

Fig. 10. Output for ex2_lra.c

The expected result on Fig. 10 confirms existence of the redundant assertion on line 17. Intuitively it means that the user should fix the other assertions first, and whenever it is done, the "redundant" assertion will hold automatically. The approach we use for removing assertions is not complete in the sense that not all dependencies between assertions are detected. However the process is sound in the sense that all dependencies are guaranteed to exist in the unwound version of the code.

Running HiFrog for the second assertion results in the output VERIFICATION FAILED and the corresponding error trace that manifests the bug.

Tuning the Strength of Summaries. Interpolation can be tuned for strength by command line parameters for propositional logic, QF_LRA and QF_UF. The parameter `--itp-algorith <algo>` specifies the interpolation algorithm for propositional logic, which is used for all theories. Variable `<algo>` ranges from 0 to 5, where the numerical values represent the propositional interpolation algorithms M_s, P, M_w, PS, PS_w, PS_s. The strength relation between the interpolation algorithms is such as M_s is the strongest, M_w is the weakest, PS_s is stronger than PS and P, and PS_w is weaker than P and PS. For more details on propositional interpolation algorithms we refer the reader to [1]. The specialized theory interpolation algorithms for QF_LRA and QF_UF can be specified, respectively, by `--itp-lra-algorithm <algo>` and `--itp-uf-algorithm <algo>`, where `<algo>` is either 0 or 2, leading to, respectively, strong and weak interpolants. For instance, verifying program `uf_interpolation.c` with the following command lines leads to summaries of different strength.

```
$ rm __summaries
$ ./hifrog --verbose-solver 2 --logic qfuf \
    --itp-algorithm 0 --itp-uf-algorithm 0 \
    --claim 1 --save-summaries strong_summaries uf_interpolation.c
$ ./hifrog --verbose-solver 2 --logic qfuf \
    --itp-algorithm 0--itp-uf-algorithm 2 \
    --claim 1 --save-summaries weak_summaries uf_interpolation.c
```

Running HiFrog with the option `--verbose-solver 2` enables printing of interpolants. Figures 11 and 12 show the interpolants generated for function `mix` (second interpolant printed by HiFrog), where the one generated with option 0 for `--itp-uf-algorithm` is strictly stronger than the one generated with option 2. These interpolants are used in the summary `c::mix#0` in the files `strong_summaries` and `weak_summaries`.

```
; Interpolant: (or (= |c::s1_out#6| |c::s2_out#6|) (or (and (= |c::s1_out#6| |c::s1_out#5|) (= |c:
:s2_out#6| |c::s2_out#5|)) (= |c::s1_out#6| |c::s2_out#6|)))
```

Fig. 11. Strong interpolant

```
; Interpolant: (or (= |c::s1_out#6| |c::s2_out#6|) (or (= |c::s1_out#6| |c::s2_out#6|) (not (and (
= |c::s1_out#5| |c::s2_out#5|) (not (= |c::s1_out#6| |c::s2_out#6|))))))
```

Fig. 12. Weak interpolant

Tuning the Size of Summaries. Proof compression directly affects the size of interpolants, and can be enabled by the command line option `--reduce-proof`. Setting `--verbose-solver 1` lists the changes that were made on the proof by the technique. For example, the following command enables proof reduction for `uf_interpolation.c` and proceeds with the interpolation afterwards.

```
# -------------------------------------
# PROOF GRAPH REDUCTION STATISTICS
# -------------------------------------
# Structural properties
# ---------------------
# Nominal num proof variables: 197
# Actual num proof variables.: 62         62
# Nodes......................: 149        131
# Nodes variation............: -12.08    %
# Leaves.....................: 63         63
# Leaves variation...........: 0.00      %
# Edges......................: 172        136
# Edges variation............: -20.93    %
# Average degree.............: 1.15       1.04
# Unary clauses..............: 57         49
# Max clause size............: 30         49
# Average clause size........: 7.62       14.79
# ------------------------
```

Fig. 13. Proof compression information

```
$ ./hifrog --verbose-solver 1 --reduce-proof \
        --logic qfuf --itp-uf-algorithm 0 --claim 1 \
        --save-summaries strong_summaries uf_interpolation.c
```

Fig. 13 shows a table from the log of `hifrog` containing the effect of proof reduction. The first column lists different types of components of a proof, such as number of variables, nodes and edges. The second column represents the corresponding statistics for the proof *before reduction*, and the third column – for the proof *after reduction*. We can see in this example that the number of nodes was reduced from 149 to 131 (12.08%), and the number of edges was reduced from 172 to 136 (20.93%).