

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. 1, consists of a function that randomly gets 1000 integers and returns their sum.

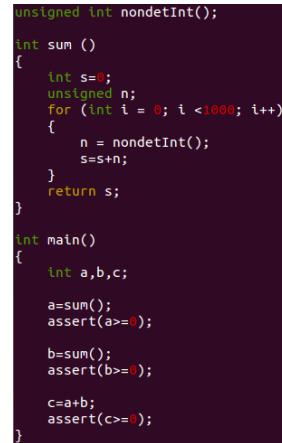
Running your claim. In this example, we add assertions to the C-program that HiFROG verifies. The C-program with three assertions is shown in Fig. 1.

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. 2 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.



```
unsigned int nondetInt();
int sum ()
{
    int s=0;
    unsigned n;
    for (int i = 0; i <1000; i++)
    {
        n = nondetInt();
        s=s+n;
    }
    return s;
}
int main()
{
    int a,b,c;
    a=sum();
    assert(a>=0);

    b=sum();
    assert(b>=0);

    c=a+b;
    assert(c>=0);
}
```

Fig. 1. `ex1_lra.c`

```

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. 2. 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. 3. `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. 3 which uses trigonometric functions and calculates $\text{Sin}^2(x) + \text{Cos}^2(x)$.

Since $\text{Sin}^2(x) + \text{Cos}^2(x) = 1$ 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. 4. Example template for `nonlin`

This command dumps a list of templates for all functions used in the program into the `__summaries` file. Fig. 4 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. 5 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. 5. 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. 6 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 complexity of multiplication encoding this turns out to be expensive.

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

```
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. 6. `ex1.uf.c`

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.

Removing redundant assertions. Fig. 7 shows a program calling a nondeterministic 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:

```
int sum1()
{
    return nondet_Int();
}

int main()
{
    int a,b,c;

    a=sum1();
    assert(a>7);

    b=sum1();
    assert(b>10);

    c=a+b;
    assert(c>7);
}
```

Fig. 7. `ex2.lra.c`

```
$ ./hifrog --claims-opt 20 ex2.lra.c
```

The expected result on Fig. 8 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.

```
Redundant assertion at:
file "ex2.lra.c",
function "main",
line 17:
c > 7
```

Fig. 8. Output for `ex2.lra.c`

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-algorithm <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 [?]. 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 9 and 10 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. 9. 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. 10. 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. 11. 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. 11 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%).