2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) | 978-1-6654-9598-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSE-Companion55297.2022.9793805

SymInfer: Inferring Numerical Invariants using Symbolic States

ThanhVu Nguyen
George Mason University
USA

KimHao Nguyen
University of Nebraska-Lincoln
USA

Hai Duong
Independent Researcher
Vietnam

ABSTRACT

We demonstrate the implementation and usage of SymInfer, a tool that automatically discovers numerical invariants using concrete and symbolic states collected from dynamic and symbolic executions. SymInfer supports expressive invariants under various forms, including nonlinear equalities, octagonal inequalities, and disjunctive min/max invariants. Experimental results show that SymInfer is effective in generating complex invariants and can often discover unknown, yet useful program properties. Video demo: **https://www.youtube.com/watch?v=VEuhJw1RBUE**.

KEYWORDS

invariant inference, symbolic execution, dynamic analysis

ACM Reference Format:
ThanhVu Nguyen, KimHao Nguyen, and Hai Duong. 2022. SymInfer: Inferring Numerical Invariants using Symbolic States. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516833

1 INTRODUCTION

Program invariants describe properties that always hold at a program location. Examples of invariants include program pre- and post-conditions, loop invariants, and assertions. Invariants are originally used to help program verification in Hoare logic but have also found uses in many other programming tasks, such as documentation, testing, debugging, code generation, and synthesis [5, 9, 10].

An important class of invariants captures numerical relations among program variables. Such *numerical invariants* can take on different mathematical forms and have various uses. Simple *linear polynomial invariants* such as $0 \leq x \leq \texttt{length}(A) - 1$ and $x \equiv y - 1$ can be used to capture out-of-bound indexing or off-by-one errors. More complex *nonlinear polynomial* relations arise in many scientific, engineering, and safety- and security-critical software [3], and can encode disjunctive information, e.g., $x^2 \leq y^2$ implies $x \leq -y \vee x \leq y$. *Max/min-plus* relations encode properties that represent a complementary form of disjunctive information, e.g., the inequality $\max(x, y) \geq 2$ is equivalent to $(x \geq y \wedge x \geq 2) \vee (x < y \wedge y \geq 2)$.

In [8, 10], we introduce SymInfer, a technique that targets the inference of rich forms of numerical invariants using *symbolic program states* captured by a symbolic execution tool. Among many benefits, symbolic states allow SymInfer to check and remove spurious invariants inferred from *concrete states* (i.e., program execution traces) by dynamic invariant generation tools, such as Daikon [5] and DIG [9]. Moreover, for many types of invariants, SymInfer can directly compute invariants over symbolic states. Our evaluation demonstrates that SymInfer establishes the state-of-the-art for inference of complex numerical invariants, especially nonlinear ones. Across benchmarks consisting of 108 challenging programs consisting of complex semantics and invariants, SymInfer is able to infer the ground truth invariants for 106 of 108 programs; the next best tool can infer only 89.

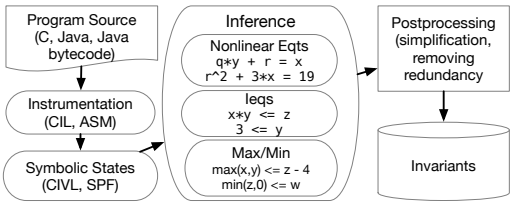

Figure 1: SymInfer Overview

The *envisioned users* for SymInfer are researchers, software engineers, and students who are interested in learning program invariants and using them in tasks such as program understanding, verification, and general program analysis. The *challenge* we address is the need for an automatic invariant generation technique and tool that accurately infer expressive invariants to capture the precise semantics of complex programs. The *methodology* we introduce is invariant analysis using a combination of symbolic and concrete states, (§2) and SMT solving and optimization. *Experimental results* on large benchmarks consisting of complex programs with nontrivial programs show the effectiveness of SymInfer [8, 10].

The algorithmic and experimental details of SymInfer are available in [8, 10]. This paper extends that work by providing details about the implementation and usage of SymInfer, which is open-source and available at **https://github.com/unsat/dig**.

2 SYMINFER

The command-line tool SymInfer takes as inputs a program written in C, Java, or Java bytecode (`.class`) marked with target locations, and returns invariants found at those locations. Fig. 1 gives an overview of SymInfer, which composes of the following phases

1. **Instrumentation**: SymInfer instruments code to obtain symbolic states using symbolic execution tool and concrete states during program execution
2. **Symbolic states collection**: SymInfer invokes a symbolic execution tool to obtain symbolic states
3. **Invariant inference**: SymInfer uses several algorithms to infer different forms of numerical invariants

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-6654-9598-1/22/05…$15.00
https://doi.org/10.1145/3510454.3516833

```
int cohendiv(int x, int y){
  assert(x >= 0 && y >= 1);
  int q=0; int r=x;
  while(r >= y){
    int a=1; int b=y;
    while[L1](r >= 2*b){
      a=2*a; b=2*b;
    }
    r=r-b; q=q+a;
  }
  [L2]
  return q;
}
```

Concrete States

x	y	a	b	q	r
15	2	1	2	0	15
15	2	2	4	0	15
15	2	1	2	4	7
⋮					
4	1	1	1	0	4
4	1	2	2	0	4
⋮					

Symbolic States

Path Conditions (Π_{L1})	**Variable Mappings** (σ_{L1})
$0 < y \wedge y \leq x$	$q \mapsto 0; r \mapsto x; a \mapsto 1; b \mapsto y$
$0 < y \wedge 2y \leq x$	$q \mapsto 0; r \mapsto x; a \mapsto 2; b \mapsto 2y$
$0 < y \wedge 2y + y \leq x < 4y$	$q \mapsto 2; r \mapsto x - 2y; a \mapsto 1; b \mapsto y$
⋮	⋮

Figure 2: Example program and concrete and symbolic states observed at location L1.

❹ **Post-processing**: SymInfer performs several simplification and filtering steps before returning invariants to user

Example. We use the C cohendiv integer division algorithm in Fig. 2 to demonstrate how SymInfer works. L1 and L2 mark the locations of interest, i.e., we want to infer the inner loop invariants at L1 and the program post-conditions at L2.

2.1 Instrumentation and States Collection

SymInfer takes as input a program with marked target locations using a special vtrace($x1, \ldots, x_n$) function and infers invariants over the variables $x_1, \ldots, x_n$. Using these vtrace calls, SymInfer instruments the program to invoke symbolic execution tool to collect symbolic states and adds "printf" statements to collect program execution traces as concrete states.

SymInfer uses symbolic execution to compute symbolic states at a considered program location L. Symbolic states consist of path conditions Π describing execution paths to L and mappings σ from program variables at L to symbolic values. Intuitively, symbolic states capture the semantics of the program at L and also compactly encode a large (potentially infinite) set of *concrete states* at L.

Example. For cohendiv, the input program is:

```
void vassume(int c);
void vtraceL1(int x, int y, in q, int r, in a, int b);
void vtraceL2(int x, int y, in q, int r);
int cohendiv(int x, int y){
   vassume(x >= 1 && y >= 1);
   ...
   while(1){
      vtraceL1(x,y,q,r,a,b); // marked location
      if (!(r >= 2*b))break;
      a=2*a; b=2*b;
   }
   ...
   vtraceL2(x,y,q,r); // marked location
   return q;
}
```

This input uses the function vassume to specify the precondition that x, y begin positive. Note the user can call vtrace over just a subset of variables to infer invariants only over those variables, e.g., vtraceL1 is called over six variables while vtraceL2 is called over only four variables. Also note how the while(c){..} loop is transformed into while(1){vtrace(..); if (!c) break; ..}. This is to capture (inductive) loop invariants, which hold the first time the loop is entered and are preserved through the loop body.

For this C program we use the symbolic execution tool CIVIL (see §3) and thus instrument code for CIVL usage as follows:

```
#include "civlc.cvh" //instr specifically for CIVL
$input int x; //instr: symbolic input
$input int y; //instr: symbolic input
//instr: collect concrete and symbolic states
void vtraceL1(int x, int y, in q, int r, in a, int b){
   printf("L1; %d; %d; %d; %d; %d; %d\n",x,y,q,r,a,b);
   $pathCondition();
}
...
int cohendiv(int x, int y){
   $assume(x >= 1 && y >= 1); //instr: assumption
   ...
   while(1){
      vtraceL1(x,y,q,r,a,b); // marked location

      if (!(r >= 2*b))break;
      a=2*a; b=2*b;
   }
   ...
   vtraceL2(x,y,q,r); // marked location
   return q;
}
```

Now SymInfer runs CIVL on the instrumented program to obtain symbolic states at the target locations indicated by vtrace. Fig. 2 shows the symbolic states at location L1 of cohendiv. As can be seen, symbolic states provide a precise logical representation of the program semantics at the target locations, and they also compactly represents a large, potentially infinite, set of concrete states. Fig. 2 also shows the concrete states at L1 when running the program on inputs (15,2) and (4,1). Notice how symbolic states encode these specific concrete states *and* those obtained when running the program on different inputs.

2.2 Invariant Inference

SymInfer uses two algorithms to infer invariants: an iterative, counterexample guided invariant refinement (CEGIR) approach to infer (potentially nonlinear) equalities and an SMT-optimization based technique to infer inequalities. The CEGIR approach uses symbolic states to check candidate equalities while the SMT approach exploits advances in constraint solving to find inequalities directly from symbolic states.

2.2.1 CEGIR. SymInfer uses a CEGIR algorithm to find *polynomial equalities* of the form $c_1t_1 + c_2t_2 + \cdots + c_nt_n = 0$, where c_i are coefficients and t_i are terms that are multiplicative combinations of relevant program variables. This algorithm iterates between two phases: *dynamic analysis* that infers candidate equalities from concrete states obtained by running the program from sample inputs and *symbolic checker* that checks candidates against the program

using symbolic states obtained from symbolic execution. If a candidate invariant is spurious, the checker also provides counterexamples. Concrete states from these counterexamples are obtained and recycled to repeat the process, and produce more accurate results. These steps of inferring and checking repeat until no new counterexamples or (true) invariants are found.

Example. We show how SymInfer find the nonlinear equalities $b = ya$ and $x = qy+r$ at location L1 in cohendiv. For demonstration purpose we only consider nonlinear equations up to degree $d = 2$.

For the six variables $\{a, b, q, r, x, y\}$ at L1, together with $d = 2$, SymInfer generates 28 terms $\{1, a, \ldots, y^2\}$ and uses them to form the template $c_1 + c_2 a + \ldots c_{28} y^2 = 0$ with 28 unknown coefficients c_i. SymInfer then collect concrete states such as those given in Fig. 2 by executing the program on random inputs and using these concrete states to form (at least) 28 linear equations. From this set of initial equations SymInfer extracts six equalities.

Now, SymInfer iteratively refines the inferred invariants. In iteration #1, SymInfer cannot refute two of these candidates $x = qy + r, b = ya$ (which are actually true invariants) and thus saves these as invariants. SymInfer finds counterexamples for the other four equalities and creates new equations from the counterexamples. SymInfer next combines the old and new equations and solves them to obtain four candidates, two of which are the already saved ones. In iteration #2, SymInfer obtains counterexamples for the 2 new candidates. With the help of the new counterexamples, SymInfer generates 3 candidates, 2 of which are the saved ones. In iteration #3, SymInfer obtains counterexamples disproving the remaining candidate and again uses the new counterexamples to generate new candidates. This time SymInfer only finds the two saved invariants $x = qy + r, b = ya$ and thus stops.

2.2.2 SMT. To infer inequalities, we previously used [8] a CEGIR approach that iterates between computing candidates from concrete states and checking them using symbolic states. In a more recent work [10], we use symbolic states to directly compute inequality relations. This approach works by first enumerate octagonal terms, such as $x - y, x + y$, and min/max-plus terms, such as $min(x, y, z)$. Then, for each term t, we use an SMT solver to compute the smallest upperbound k for t from symbolic states. If k is found, we obtain the candidate invariant $t \leq k$. Otherwise (i.e., if k is ∞ or cannot be determined), we discard the relation $t \leq k$.

Similarly, we also compute the largest lower-bound k' to obtain the inequality $k' \leq t$. This approach leverages the power of modern constraint solvers, which, in addition to finding satisfiability assignments, can find *optimal* assignments with respect to objective constraints using linear optimization techniques.

Example. For the six variables at L1, SymInfer enumerates $\binom{6}{2}$ pairs of variables $(a, b), (a, q), \ldots (x, y)$ and for each pair forms eights terms involving at most two variables such as $\{a, b, \ldots, -a - b, a + b\}$. Then, for each term t SymInfer computes the smallest upperbound k and the largest lower-bound k' to form the invariant $k' \leq t \leq k$. For example, for the term $-a - y$, SymInfer infers $-\infty \leq -a - y \leq -2$, which simplifies to $2 \leq a + y$ (the input y is assumed to be ≥ 1 but has no upperbound and a is initialized with 1 and always doubled).

Similarly, to infer min- and max-plus invariants such as $k' \leq min(x, y) \leq k$, SymInfer performs the same process of generating terms and computing upper- and lower-bounds. In this example, SymInfer found several such invariants, however, our post-processing step determined that they are weaker than the other obtained equalities and inequalities and therefore removed them.

2.3 Post-Processing and Invariant Results

Depending on the number of variables and form of invariants, SymInfer could generate many invariants (e.g., each octagonal and max/min term can produce an invariant candidate). SymInfer uses a post-processing step, which consists of two parts, to reduce the number of reported invariants. The first part simply checks generated invariants against all cached concrete states and removes violated ones. This part is efficient (we simply instantiate and check candidate relations with concrete values), but removes few results (because most generated invariants are already valid). The second part removes redundant invariants. From a set of candidate invariants, we extract a subset of *independent* relations and check if every member of the set is not implied by other relations in that set. This part is more time-consuming, but effective in reducing many inequalities to just a few strongest and relevant ones–making it much easier for the user to analyze and use the reported results.

Example. For the cohendiv program, SymInfer got 272 invariants (4 equations, 41 inequalities, and 227 min/max) over the two locations L1 and L2. After post-processing, the number of invariants reduced to just 15 (8 at L1 and 7 at L2).

At the end, SymInfer returns at L1 (loop) invariants such as

$$x = qy + r; \quad ay = b; \quad b \leq x; \quad y \leq r; \quad 0 \leq q; \quad 1 \leq b; \quad 1 \leq y$$

and at L2 the (post-condition) invariants such as

$$x = qy + r; \quad r \leq y - 1; \quad 0 \leq r; \quad r \leq x$$

These relations are sufficiently strong to understand the semantics and verify the correctness of cohendiv. The key invariant is the nonlinear equality $x = qy + r$, which captures the precise behavior of integer division: the dividend x equals the divisor y times the quotient q plus the remainder r. The other inequalities also provide useful information. For example, the invariants at the program exit reveal several required properties of the remainder r, e.g., non-negative ($0 \leq r$), at most the dividend ($r \leq x$), but strictly less than the divisor ($r \leq y - 1$).

3 DESIGN AND TOOL USAGE

SymInfer is implemented in Python and uses SymPy for equation solving (to infer equalities) and represent numerical relations. SymInfer uses different instrumentation and symbolic execution tools depending on the input program. For C, we use CIL [7] for instrumentation and CIVL [11] for symbolic execution. For Java and Java bytecode, we use the ASM library [2] for instrumentation and Symbolic PathFinder [1] for symbolic execution. SymInfer uses the Z3 SMT solver [4] to check and produce models representing counterexamples. Z3 is also used to identify and remove redundant invariants in post-processing.

3.1 Design

SymInfer has several designs to aid development and adoption. In particular, SymInfer is designed to be configurable, extendable, and take advantage of multicore (parallel) processing,

Configurability. SymInfer's is highly configurable and contains more than 30 settings allowing the user to control how the tool works. By default, these settings are chosen to allow SymInfer to work with a wide-range of benchmark programs, e.g., from those with few simple linear invariants to those with complex nonlinear relations involving dozen of variables. The user can change these settings using command-line options or the `settings.py` file.

Several useful settings include: `-maxdepth d` (generate invariants only up to degree d and can significantly speed up SymInfer); `-nominmax/-noieqs/-noeqts` (do not generate certain forms of invariants); `-se_min/maxdepth` (controls the depth of symbolic execution); `-noss` (do *not* use symbolic states and performs pure dynamic invariant generation over randomly generated inputs); `-uterms "t1; t2; .."` (infer invariants involving terms representing specific, complex information, e.g., $t1 = 2^x, t2 = \log(n)$).

Extensibility. SymInfer is designed to be modular, allowing the user to easily extend it to support new form of invariants. To support new invariants, the user just needs to extend the `Invariant` and `Inference` abstract classes and override abstract methods such as `infer check` to take advantage of SymInfer's current CEGIR and SMT-based algorithms, post-processing, parallel computing, etc.

Multicore Processing. SymInfer leverages the increasingly popular and affordable multicore architecture. The tool performs many independent tasks in parallel, e.g., running symbolic execution, generating invariants at different locations, computing upper bounds for terms, and checking candidate invariants. Parallel processing is crucial to the performance of SymInfer as it allows the tool to process and analyze thousands of candidate invariants at multiple program locations simultaneously.

Usage. The Github repository in §1 provides detailed instructions for obtaining, building, and running SymInfer. The easiest way to try SymInfer is through the provided Dockerfile, but the user can also build SymInfer directly from source. SymInfer is designed to be integrated easily with other projects and tools. The user can call SymInfer as a blackbox or use its Python API to infer invariants (e.g., the Dynamite project [6] calls SymInfer as a blackbox to infer invariants program termination and non-termination analyses).

3.2 Run Output

Fig. 3 shows the results of running SymInfer on the cohendiv program on a 64-core AMD CPU 4 GHZ Linux system with 64 GB of RAM. Here, SymInfer got 45 symbolic states for the two target locations in 4.7s. Next, we got 41 inequality, 227 min/max, and 4 equality invariants. After reprocessing, SymInfer reduced the number of invariants from 272 to just 15 invariants.

We also see that the *real* (wall clock) time of the entire process is just 20.89s even though accumulative time spent by all CPU is 368.18 seconds. This shows that SymInfer is effective in exploiting multicore processing (i.e., without using multicore, the run time would be 6 mins instead of just 21s).

```
# time python3 -O syminfer.py ../tests/cohendiv.c -log 3
alg:INFO:analyzing '../tests/cohendiv.c'
alg:INFO:got 45 symstates at 2 locs in 4.69s
alg:INFO:got 41 ieqs in 0.65s
alg:INFO:got 227 minmax in 1.27s
alg:INFO:got 4 eqts in 13.24s
alg:INFO:check 272 invs using 456 traces (0.25s)
alg:INFO:simplify 272 invs (2.23s)
vtraceL1(8 invs): a*y - b == 0; q*y + r - x == 0; -q <= 0;
a - b <= 0; r - x <= 0; b - r <= 0; -b + y <= 0; -a - y <= -2
vtraceL2(7 invs): q*y + r - x == 0; -q <= 0; -r <= 0;
r - x <= 0; q - x <= 0; r - y <= -1; -q - x <= -1
----------------------------------------
Time: real 20.89 secs; usr 368.18 secs
```

Figure 3: Running SymInfer

4 EVALUATION

We evaluate SymInfers [10] using four benchmark suites consisting of 108 programs. These programs come with known or documented invariants, which we use as ground truths for comparison.

Our experiments show that SymInfer is able to infer the ground truth invariants for 106 of 108 programs; the next best tool can infer only 89. In many cases, SymInfer found undocumented but interesting invariants revealing useful facts about program semantics and complexity bounds. The ability to exploit and reuse symbolic states allows SymInfer to strike a balance between expressive power and computational cost, while guaranteeing correctness, to establish state-of-the-art performance in numerical invariant inference.

ACKNOWLEDGMENT

We thank the anonymous reviewers for helpful comments. This material is based in part upon work supported by the National Science Foundation under grant numbers 1948536, 2107035 and U.S. Army Research Office under grant number W911NF-19-1-0054.

REFERENCES

[1] Saswat Anand, Corina S Păsăreanu, and Willem Visser. 2007. JPF–SE: A symbolic execution extension to Java Pathfinder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 134–138.
[2] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).
[3] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The Astrée analyzer. In *European Symposium on Programming*. Springer, 21–30.
[4] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[5] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.
[6] TonChanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: dynamic termination and non-termination proofs. *PACMPL* 4, OOPSLA (2020), 1–30.
[7] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*. Springer, 213–228.
[8] ThanhVu Nguyen, Matthew Dwyer, and William Visser. 2017. SymInfer: Inferring Program Invariants using Symbolic States. In *Automated Software Engineering*. IEEE, 804–814.
[9] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *International Conference on Software Engineering*. IEEE, 683–693.

[10] Thanhvu Nguyen, KimHao Nguyen, and Matthew Dwyer. 2021. Using Symbolic States to Infer Numerical Invariants. *Transactions on Software Engineering (TSE)* (2021).

[11] Stephen F Siegel, Manchun Zheng, Ziqing Luo, Timothy K Zirkel, Andre V Marianiello, John G Edenhofner, Matthew B Dwyer, and Michael S Rogers. 2015. CIVL: the concurrency intermediate verification language. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.