Generalizing Specific-Instance Interpolation Proofs with SyGuS

Muqsit Azeem
TCS Research
Pune, India
muqsit.azeem@tcs.com

Kumar Madhukar
TCS Research
Pune, India
kumar.madhukar@tcs.com

R Venkatesh
TCS Research
Pune, India
r.venky@tcs.com

ABSTRACT

Proving correctness[1] of programs is a challenging task, and consequently has been the focus of a lot of research. One way to break this problem down is to look at one execution path of the program, argue for its correctness, and see if the argument extends to the entire program. However, that may not often be the case, i.e. the proof of a given instance can be overly specific. In this paper, we propose a technique to generalize from such specific-instance proofs, to derive a correctness argument for the entire program. The individual proofs are obtained from an off-the-shelf interpolating prover, and we use Syntax-Guided Synthesis (SyGuS) to generalize the facts that constitute those proofs. Our initial experiment with a prototype tool shows that there is a lot of scope to guide the generalization engine to converge to a proof very quickly.

CCS CONCEPTS

• **Theory of computation → Invariants**; • **Software and its engineering → Software verification**; • **Computing methodologies →** *Instance-based learning*;

KEYWORDS

Verification, Interpolants, Syntax-Guided Synthesis

ACM Reference Format:
Muqsit Azeem, Kumar Madhukar, and R Venkatesh. 2018. Generalizing Specific-Instance Interpolation Proofs with SyGuS. In *ICSE-NIER'18: 40th International Conference on Software Engineering: New Ideas and Emerging Results Track, May 27-June 3 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183399.3183412

1 INTRODUCTION

Computer programs may be regarded as formal mathematical objects whose properties are subject to mathematical proof [5]. However, proving properties of programs is a challenging task in practice, to say the least. The problem has worsened due to the increasing complexity of real-world programs that are used to synthesize modern day hardware and software. Consequently, even the state-of-the-art verification tools and techniques fail to analyze the correctness of an entire program, i.e. the set of all its possible executions/behaviors. It becomes natural to ask if one could i) analyze a subset of the program executions, and ii) extend the correctness argument for this subset, to the entire program. This paper proposes an approach to do this automatically.

Given a program, one may isolate a subset of behaviors, i.e. find an under-approximation, by fixing an unwinding for the loops. For example, consider the program in Fig. 1 (a), and an under-approximation of it in Fig. 1 (b) obtained with an unwinding of 2. Like in this example, we assume that the property to be verified is given as an assertion in the program. The under-approximate program can be encoded as a logical formula, by transforming the statements into the *static single assignment (SSA)* form, and given to a theorem prover, e.g. Z3 [13]. The theorem prover may return a counterexample to the correctness of the under-approximate program (in which case the original program is indeed unsafe), or a proof that this (subset of behaviors) does not violate the property. Such resolution proofs, however, are not quite usable directly.

One possible way to extract meaningful information from the resolution proofs is to generate interpolants [12, 17] from them, using an interpolating theorem prover [21], e.g. CSIsat [4] or iZ3 [23]. Informally, an interpolant is a predicate that separates good program states from bad ones, and thus helps in forming program proofs. In particular, in our case, they can be used to construct a Floyd-Hoare style proof of correctness of the under-approximate program. While the interpolating provers have the ability to focus on relevant facts, in most cases these facts are very specific to the under-approximate program. As a result, the proof does not hold for the original program. Nevertheless, these specific interpolants are very often instantiations of more "general" facts, that might be useful in establishing the correctness of the entire program at once. Thus, this limitation of interpolating provers furnishing overly specific proofs can be worked around by generalizing from several such proofs.

Syntax-Guided Synthesis (SyGuS) [2] is a recently proposed framework for the program synthesis problem. An important aspect of this framework is its ability to generate specifications, in a given collection of templates (or, a grammar), from a set of examples. We use a SyGuS solver to generalize the facts (interpolants) obtained from interpolation proofs. The input to the solver is these facts, and a grammar that specifies the template in which a generalized fact is to be obtained. Employing an external generalization mechanism like this allows us to use any off-the-shelf interpolating prover, irrespective of whether the prover itself focuses on general facts or not.

Our approach involves performing the following steps repeatedly, till either a counterexample or a proof is obtained:

- constructing under-approximations of the original program

[1] Throughout this document, correctness refers to *partial correctness* of programs. We do not consider the question of *termination* in this work.

ICSE-NIER'18, May 27-June 3 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5662-6/18/05…$15.00
https://doi.org/10.1145/3183399.3183412

- generating proofs of their correctness, using an interpolating prover (or returning *unsafe* if a counterexample is found)
- using a SyGuS solver to generalize the facts, and strengthening/refining them as required.

Note that this algorithm may not converge in every case, as program verification is undecidable in general.

The core contribution of this paper is a novel technique that combines interpolation and SyGuS, for generalizing specific-instance proofs into a proof of the entire program.

Outline of the paper We start with an illustrative example in the next section, followed by a discussion on the related work in Section 3. Section 4 presents the core elements of our approach. We conclude in Section 5, and lists interesting directions of pursuing this further.

2 ILLUSTRATIVE EXAMPLE

Consider the program shown in Fig. 1 (a). The variables x and y are assigned 0 initially, and are then incremented in a loop an unknown number of times. Then, in another loop, both the variables are decremented as long as both x and y are non-zero. At this point, it is asserted that both the variables are zero. This is a safe program, i.e. there does not exist a feasible path from the initial state to an assertion violating state.

```
int x = 0;
int y = 0;

while(*){
  x++; y++;
}

while(x != 0 && y != 0){
  x--; y--;
}

assert(x == 0 && y == 0);
```

(a)

```
int x = 0;
int y = 0;

x++; y++;
x++; y++;

assume(x != 0 && y != 0);
x--; y--;

assume(x != 0 && y != 0);
x--; y--;

assume(x == 0 || y == 0);
assert(x == 0 && y == 0);
```

(b)

Figure 1: Example program (a), and a specific path in it with an unwinding of 2 (b)

Let us consider an execution path of this program, shown in Fig. 1 (b), that goes through both the while loops twice. We start by looking for a proof of correctness of this specific instance. Our approach is to transform the program into SSA form, encode the statements and the negated assertion as a logical formula, and see if the formula is satisfiable. This can be done using an interpolating theorem prover, e.g. iZ3. When invoked with the constraints shown in Fig. 2 (a), iZ3 return a set of interpolants as the `unsat` proof (see Fig. 2 (b)).

While we did obtain a proof of correctness of the specific case, this particular proof is not of much interest to us. The reason being that this proof does not *generalize*. In other words, it uses facts that can only help in proving this instance correct (see the proof in Fig. 3). These facts are not relevant to the correctness of the entire program, or even for a different instance of the program, e.g. when both the while loops are executed thrice. This happens with every instance of the program in Fig. 1 (a) that we attempt to prove and we end up with a diverging set of proofs. Instead, what we wish to obtain is a general proof that works for every instance, i.e. a proof of the entire program.

```
(compute-interpolant
  (and (= x0 0) (= y0 0))
  (and (= x1 (+ x0 1)) (= y1 (+ y0 1)))
  (and (= x2 (+ x1 1)) (= y2 (+ y1 1)))
  (and (not (= x2 0)) (not (= y2 0)))
  (and (= x3 (- x2 1)) (= y3 (- y2 1)))
  (and (not (= x3 0)) (not (= y3 0)))
  (and (= x4 (- x3 1)) (= y4 (- y3 1)))
  (or (= x4 0) (= y4 0))
  (not (and (= x4 0) (= y4 0))))
```

(a)

```
unsat
(and (= 0 y0) (= 0 x0))
(and (= 1 y1) (= 1 x1))
(and (= 2 y2) (= 2 x2))
(and (= 2 y2) (= 2 x2))
(and (= 1 y3) (= 1 x3))
(and (= 1 y3) (= 1 x3))
(and (= 0 y4) (= 0 x4))
(and (= 0 y4) (= 0 x4))
```

(b)

Figure 2: Input formula for the interpolating theorem prover iZ3 (a), and the interpolants obtained as unsat proof (b)

```
{True}                        x = 0; y = 0;          {(x = 0) && (y = 0)}

{(x = 0) && (y = 0)}            x++; y++;            {(x = 1) && (y = 1)}

{(x = 1) && (y = 1)}            x++; y++;            {(x = 2) && (y = 2)}

{(x = 2) && (y = 2)}      assume((x != 0) &&
                                (y != 0));            {(x = 2) && (y = 2)}

{(x = 2) && (y = 2)}            x--; y--;            {(x = 1) && (y = 1)}

{(x = 1) && (y = 1)}      assume((x != 0) &&
                                (y != 0));            {(x = 1) && (y = 1)}

{(x = 1) && (y = 1)}            x--; y--;            {(x = 0) && (y = 0)}

{(x = 0) && (y = 0)}      assume((x == 0) ||
                                (y == 0));            {(x = 0) && (y = 0)}

{(x = 0) && (y = 0)}      assert(not((x == 0)
                             && (y == 0)));           {False}
```

Figure 3: Floyd-Hoare proof of correctness of the program in Fig. 1 (b), obtained from the interpolants shown in Fig. 2 (b)

To obtain a general proof of correctness, we attempt to generalize the facts in the seemingly divergent set of proofs. This generalization is performed using a Syntax-Guided Synthesis tool, e.g. CVC4-1.5 [25] or EUSolver [3]. Consider the facts available just before the `while` loops in the program as shown in Fig. 1 (a). For an unwind of 2, the facts at the loop-head of the second loop are: {(x = 2) ∧ (y = 2)}, {(x = 1) ∧ (y = 1)} and {(x = 0) ∧ (y = 0)}. These facts are given as input to the SyGuS solver (CVC4-1.5, in our case) in order to obtain another fact, in a pre-specified syntax (grammar), that *generalizes* the input facts.

This does not turn out to be the case, however. The reason being that we only have *positive examples* to begin with. In other words, we are asking the SyGuS solver to return a fact that is consistent with {(x = 2) ∧ (y = 2)}, {(x = 1) ∧ (y = 1)} and {(x = 0) ∧ (y = 0)}. The solver may return a trivial answer, e.g. {x = x}. This mandates a refinement step therefore. We perform the refinement in a counterexample guided fashion, where the counterexamples act as *negative examples* during the synthesis.

Obtaining counterexamples at this stage is easy, given the (trivial) invariant is insufficient to prove the property. We use Cbmc [11], a bounded model checker for C programs, for this purpose. The input to Cbmc is a modification of the original program - all loops are replaced by non-deterministic assignments to the variables modified within that loop, under the constraint that the fact returned by the SyGuS solver holds at the corresponding program point. Cbmc returns a counterexample: {`x = 268435456` ∧ `y = 0`}, showing insufficiency of the invariant {`x = x`} in proving the property.

This counterexample is given to the SyGuS solver, along with the other facts. Now, we ask the solver to return a fact that is consistent with {(`x = 2`) ∧ (`y = 2`)}, {(`x = 1`) ∧ (`y = 1`)} and {(`x = 0`) ∧ (`y = 0`)}, but inconsistent with {`x = 268435456` ∧ `y = 0`}. This solver returns {`x ≤ y`}, and the process is repeated. This gives us one more negative example, and the SyGuS solver converges to the desired invariant, {`x = y`}, in the third iteration.

It must be noted that this refinement process may itself diverge, with a new counterexample each time. It is therefore important to switch back and forth between adding positive examples (more proofs, obtained from different under-approximations), and negative ones (more counterexamples). There are several heuristics that can be used to guide this search. We are experimenting with them, and we plan to publish these heuristics along with their performance results soon.

Once a strong enough invariant is obtained for the second loop, we shrink the original program by replacing the loop with an assertion that says the invariant must hold. And then we repeat this till we find an invariant for the first loop, which is {`x = y`} again in this case. A proof based on this fact indeed generalizes every specific instance proof that was obtained from the interpolating prover.

3 RELATED WORK

The novelty of our work lies in connecting interpolation and generalization, using SyGuS. Since these techniques have been well-studied in various different contexts, we can only give a brief overview of the vast amount of relevant literature.

Interpolants have been used in a variety of applications in computer science, ranging from realizability of specifications [7], to formal verification of software [17, 22] and hardware systems [20]. Albarghouthi et al. [1] look at proofs of bounded collection of program executions to derive an invariant using Craig interpolation methods [10, 19, 21]. Their approach is based on the heuristic that simpler proofs are more likely to generalize, i.e. their focus is on an interpolant generator that constructs simple interpolants (or, facts). Instead, our approach takes the interpolants [23] generated from off-the-shelf provers as is, but uses syntax guided synthesis to derive conjectured invariants in a syntactic template (a *grammar*).

Generalization has been used, for both proofs and counterexamples, to tackle the scalability issue in verification of large systems. For example, in IC3 [6], a counterexample to induction is generalized in order to refine over-approximations of sets of reachable states [15]. In a recently proposed interactive system, Ivy [24], users guide the system to generalize such counterexamples to induction, till an inductive invariant is found. This is not just limited to counterexamples. Heizmann et al. [16] look at correctness arguments and then generalize, as an automaton, the set of executions for which the same argument applies. This is close to a compositional approach where one may argue about program correctness by separately arguing about subsets of executions that cover the entire program. In contrast, we attempt to generalize the argument itself so that it becomes applicable to the program in general.

Syntax Guided Synthesis has received a lot of attention in the recent years, as a useful framework for the program synthesis problem. This has led to advancements, in research as well as in tools [3, 14, 25]. Our approach directly benefits from these - an efficient tool helps lead to quicker convergence during the generalization step.

4 OUR APPROACH

Our technique utilizes the ability of interpolating provers to focus on "relevant" facts, and that of SyGuS solvers to generalize the specific-instance facts into a general proof. The important elements of our approach are as follows.

4.1 Under-approximation

We consider under-approximations of the input program and look for correctness arguments only for those paths that are there in the under-approximation. These under-approximations may be chosen such that the theorem prover comes up with a proof quickly. Limiting the loop unwindings (e.g. as in our illustrative example, Fig. 1 (b)), is one way to restrict the set of program behaviors. However, other under-approximations would work equally well. For instance, one may add *assume(condition)* statements in the program to disallow certain paths. A program execution goes past an assume statement only if *condition* evaluates to true at that point.

4.2 Interpolation Proofs

We use interpolating provers (e.g. iZ3 [23], MathSAT 5 [9], SMTInterpol [8], CSIsat [4]) to prove correctness of under-approximate program, or to refute them with a counterexample. By obtaining interpolants from proofs, we focus on facts that are relevant and avoid deducing information that is irrelevant to the analysis. We transform the under-approximate program into SSA form, and construct a logical formula that encodes program statements and the negation of the assertion. The prover either returns `unsat`, followed by a *refutation proof* (see Fig. 2 (b)) that establishes correctness of the under-approximated program, or it may return `sat`, which means that the original program is *unsafe*. The refutation proof may involve facts that are general, although that is infrequently the case. In most cases, a generalization is needed as the proofs are overly specific to the input instance.

4.3 Generalization

The generalization is done in our approach with the help of a SyGuS solver. The synthesis problems that we solve takes as input positive examples (the proofs obtained from interpolating provers), negative examples (counterexamples obtained from Cbmc), and a grammar that defines the set of possible templates for the general fact. In the absence of negative examples, the solver may return a trivial invariant as shown in the illustrative example. It is noteworthy that a relevant generalization may help the overall proof converge rapidly, whereas an irrelevant generalization may result in explosion of the proof search, or even divergence. Therefore, one needs to carefully design a set of heuristics to guide the generalization engine to converge quickly.

4.4 Refinement or Strengthening

A refinement is invariably needed in our approach, and is guided by a set of heuristics. At each step of generalization, the technique needs to decide if *a*) a positive example should be added to the synthesis problem given to SyGuS (by adding proofs of new under-approximations), *b*) a negative example should be added (using Cbmc to produce a counterexample, showing insufficiency of the invariant synthesized so far), or *c*) a strengthening is required, i.e. the positive examples (proofs) need to be enriched with additional facts about other program variables. For example, the invariant {x = y} cannot be obtained if the proofs contain facts about y alone.

We have built a prototype tool that implements our idea, and have tested it on several sample programs. The initial results look promising. At the link *https://www.cmi.ac.in/~madhukar/sein/* we present artifacts that describe, step-by-step, how the tool performs on selected examples.

5 CONCLUSION AND FUTURE WORK

Our work integrates two well-studied techniques - interpolation and syntax based generalization - to verify programs with respect to a given assertion. It starts with under-approximations of the original program and attempts to prove them safe. If a counterexample is found, then the original program is reported unsafe. However, if a proof is obtained, a generalization is attempted to extend the proof to the original program. The crucial aspect of our approach is that the techniques complement one another - interpolation has the ability to focus on what is important, but it can get overly specific, whereas generalization with SyGuS can exactly remove this specificity.

In the immediate future, we plan to develop our prototype tool into a mature software, and add several heuristics to make it efficient. It would also be interesting to see if the generalizer could be guided by an oracle, instead of examples, as in Oracle-Guided Synthesis [18]. We would also like to see if SyGuS itself could be tweaked for our purpose. For example, by (*minimally*) refining the grammar automatically, if the tool is struggling to furnish an invariant in the given grammar. We plan to take up these research directions as we go ahead.

REFERENCES

[1] Aws Albarghouthi and Kenneth L. McMillan. 2013. Beautiful Interpolants. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*. Springer-Verlag, Berlin, Heidelberg, 313–329. https://doi.org/10.1007/978-3-642-39799-8_22

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 1–8.

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Axel Legay and Tiziana Margaria (Eds.), Vol. 10205. 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

[4] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. 2008. *CSIsat: Interpolation for LA+EUF.* Springer Berlin Heidelberg, 304–308.

[5] Robert S. Boyer and J. Strother Moore. 1985. Program Verification. *J. Autom. Reasoning* 1, 1 (1985), 17–23.

[6] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science)*, Ranjit Jhala and David A. Schmidt (Eds.), Vol. 6538. Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[7] Davide G. Cavezza and Dalal Alrajeh. 2017. Interpolation-Based GR(1) Assumptions Refinement. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 281–297.

[8] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings.* 248–254.

[9] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. *The MathSAT5 SMT Solver.* Springer Berlin Heidelberg, 93–107.

[10] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. 2009. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. *CoRR* abs/0906.4492 (2009).

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems.* LNCS, Vol. 2988. Springer, 168–176.

[12] William Craig. 1957. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *J. Symb. Log.* 22(3) (1957), 269–285.

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[14] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 499–512.

[15] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. 2013. Better generalization in IC3. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013.* 157–164.

[16] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[17] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions from Proofs. *SIGPLAN Not.* 39(1) (Jan. 2004), 232–244.

[18] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 215–224.

[19] Daniel Kroening and Georg Weissenbacher. 2007. Lifting Propositional Interpolants to the Word-Level. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD '07)*. IEEE Computer Society, Washington, DC, USA, 85–89. https://doi.org/10.1109/FMCAD.2007.31

[20] Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (LNCS)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.), Vol. 2725. Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[21] K. L. McMillan. 2005. An Interpolating Theorem Prover. *Theor. Comput. Sci.* 345, 1 (Nov. 2005), 101–121. https://doi.org/10.1016/j.tcs.2005.07.003

[22] Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. Springer-Verlag, Berlin, Heidelberg, 123–136. https://doi.org/10.1007/11817963_14

[23] Kenneth L. McMillan. 2011. Interpolants from Z3 Proofs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD '11)*. FMCAD Inc, Austin, TX, 19–27.

[24] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 614–630. https://doi.org/10.1145/2908080.2908118

[25] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9207. Springer, 198–216. https://doi.org/10.1007/978-3-319-21668-3_12