Synthesis through Unification Genetic Programming

Thomas Welsch
Computer Science, University of Liverpool, UK

Vitaliy Kurlin
Computer Science, University of Liverpool, UK

ABSTRACT

We present a new method, Synthesis through Unification Genetic Programming (STUN GP), which synthesizes provably correct programs using a Divide and Conquer approach. This method first splits the input space by undergoing a discovery phase that uses Counterexample-Driven Genetic Programming (CDGP) to identify a set of programs that are provably correct under unknown unification constraints. The STUN GP method then computes these restraints by synthesizing predicates with CDGP that strictly map inputs to programs where the output will be correct.

This method builds on previous work towards applying Genetic Programming (GP) to Syntax Guided Synthesis (SyGus) problems that seek to synthesize programs adhering to a formal specification rather than a fixed set of input-output examples. We show that our method is more scalable than previous CDGP variants, solving several benchmarks from the SyGus Competition that cannot be solved by CDGP. STUN GP significantly cuts into the gap between GP and state-of-the-art SyGus solvers and further demonstrates Genetic Programming's potential for Program Synthesis.

CCS CONCEPTS

• **Software and its engineering** → **Genetic programming**; • **Theory of computation** → *Program verification*;

KEYWORDS

Genetic programming, Search Based Program Synthesis, Divide and Conquer

ACM Reference Format:
Thomas Welsch and Vitaliy Kurlin. 2020. Synthesis through Unification Genetic Programming. In *Genetic and Evolutionary Computation Conference (GECCO '20), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3377930.3390208

1 INTRODUCTION

Program Synthesis is the problem of synthesizing a program given a specification dictating the program's behavior. A prominent variant of Program Synthesis is Syntax Guided Synthesis (SyGus). [1] SyGus imposes a formal specification *Spec* consisting of logical constraints on the input and output of the program to be synthesized. *Spec* is supplemented by a Grammar G, a set of production rules from which the program must be constructed. An example of a SyGus formulation of the max4 problem is given by the combination of Figure 1 containing a grammar G for the Conditional Linear Integer Arithmetic (CLIA) domain and Figure 2, containing a formal specification *Spec*. The key property of SyGus formulations is that they allow for the creation of provably correct programs with respect to *Spec* found through search based methods on the syntactic space restricted by G.

Figure 1: A grammar for Conditional Linear Integer Arithmetic (CLIA).

```
I ::= I + I | I - I | ite(B,I,I) | -1 | 0 | 1
| v1 | v2 | ... | vn
B ::= and(B,B) | or(B,B) | not(B) | B = B
| I < I | I <= I | I = I | I >= I | I > I
```

Standard Genetic Programming (GP) and Grammatical Evolution (GE) cannot be applied to the SyGus problem as, while they can construct programs from G [13], they require input-output examples to inductively synthesize a solution. Counterexample-Driven Genetic Programming (*CDGP*), however, demonstrated that through interaction with a Satisfiability Modulo Theorem (SMT) Solver a set of examples could be obtained that formed a suitable fitness gradient for GP. Consequently CDGP is capable of producing provably correct programs on SyGus problems. [11]

CDGP supports grammars for two domains: Strings with Linear Integer Arithmetic (SLIA) and Conditional Linear Integer Arithmetic (CLIA). Problems in the SLIA domain seek to manipulate strings, usually to match some specific syntax, while CLIA problems result in the construction of tree-like programs where inner nodes consist of boolean predicates that form paths to leaves consisting of terms that return an integer. The paper [7] demonstrated that CDGP is competitive with the state-of-the-art SyGus solver CVC4 [6] in the SLIA domain and in fact is superior on purely declarative specifications that are not supplemented with input-output examples. CDGP was less competitive, however, on problems in the CLIA domain. We posit that the reason for this lies in CDGP's approach to CLIA problems. CDGP implicitly solves CLIA problems by simultaneously discovering terms which are correct on a subset of the problem's input space, predicates which map to these terms, and a unification of these terms and predicates that form a correct program. CDGP's approach is sound but as arity of the function to synthesize increases performing these tasks simultaneously

GECCO '20, July 8–12, 2020, Cancún, Mexico

ACM ISBN 978-1-4503-7128-5/20/07…$15.00
https://doi.org/10.1145/3377930.3390208

```
(set-logic LIA)
;;Specifies structure of target function,
;;has input Ints x,y,z,w and the
;;function returns an Int.
(synth-fun max4 ((x Int) (y Int) (z Int)
(w Int)) Int)

;;Declaration of variables.
(declare-var x Int) (declare-var y Int)
(declare-var z Int) (declare-var w Int)

;;Constraints target function must satisfy.
(constraint (>= (max4 x y z w) x))
(constraint (>= (max4 x y z w) y))
(constraint (>= (max4 x y z w) z))
(constraint (>= (max4 x y z w) w))
(constraint (or (= x (max4 x y z w))
            (or (= y (max4 x y z w))
            (or (= z (max4 x y z w))
        (= w (max4 x y z w))))))
(check-synth)
```

Figure 2: A declarative specification for the max4 problem.

becomes harder due to the curse of dimensionality. Consequently, CDGP struggles to synthesize programs with higher function arities. [7]

Our new method, Synthesis through Unification GP (STUN GP), exploits properties of the CLIA domain to split the tasks of term discovery, predicate mapping and unification into separate processes. This decomposes the problem into sub-problems of lower dimensionality allowing the algorithm to cope better with increases in arity. We demonstrate in our results section that on canonical SyGus benchmarks an implementation of STUN GP solves problems up to an arity of 15, a nearly three fold increase upon the max arity previously achieved by CDGP. This significantly closes the gap between GP and state-of-the-art SyGus solvers in the CLIA domain. The source code for this project is available at https://github.com/twelsch1/STUN_GP.

2 RELATED WORKS AND BACKGROUND THEORY

Program synthesis has traditionally been an interest of the formal methods community. "Search Based Program Synthesis" gives a good introduction of how this problem has been considered historically and how the new Search Based Program Synthesis paradigm encompassing SyGus emerged. [5]

Recently attention in the GP community has begun to turn towards Program Synthesis, as noted by [12], and GP inherently performs a form of Program Synthesis for any given problem. However, the only other GP method we are aware of capable of solving SyGus problems is CDGP. Consequently, CDGP has a strong influence on and is indeed integral to our approach.

```
(set-logic LIA)
(set-option :produce-models true)
;;Function is now defined
;;as synthesized program.
(define-fun max4
((x Int)(y Int)(z Int)(w Int))
Int (ite (and (<= w x) (= y x))
(mod y (- 9)) z))

(declare-fun x () Int) (declare-fun y () Int)
(declare-fun z () Int) (declare-fun w () Int)
;;Assert that the program defined
;;does NOT satisfy max4.
(assert (not (and (>= (max4 x y z w) x)
  (>= (max4 x y z w) y)
  (>= (max4 x y z w) z)
  (>= (max4 x y z w) w)
  (or (= x (max4 x y z w))
  (or (= y (max4 x y z w))
  (or (= z (max4 x y z w))
  (= w (max4 x y z w))))))))

;;Check for satisfiability,
;;get counterexample if program incorrect.
(check-sat)
(get-value (x y z w))
```

Figure 3: An example of a query that verifies a single program.

We give special consideration to two concepts from the formal methods community: Counterexample Guided Inductive Synthesis (CEGIS) [1] and Synthesis through Unification (STUN) [2]. CDGP can be thought of as an extension of CEGIS into the GP domain as it uses a CEGIS like workflow to construct a training set. Our new method, meanwhile, can be thought of as an extension of STUN into the GP domain.

2.1 Counterexample Guided Inductive Synthesis

Counterexample Guided Inductive Synthesis (CEGIS) is an algorithm that synthesizes a program through iterative interaction with a Satisfiability Modulo Theorem (SMT) Solver. The Learning Algorithm, or Synthesizer, produces a best guess program based off its knowledge of the search space and presents this to the SMT Solver. The solver then formally verifies the program to determine if it is correct. If it is correct the Solver communicates this and the algorithm terminates and returns the provably correct program. If it is not correct the Solver returns a counterexample which the Learning Algorithm then uses to try and produce a better candidate. [1]

2.2 Counterexample-Driven Genetic Programming (CDGP)

Counterexample-Driven Genetic Programming (CDGP) is an extension of Genetic Programming that can synthesize

provably correct programs from formal specifications. CDGP is in most respects a standard implementation of GP. Programs are represented as standard tree structures, evolution is achieved through crossover and mutation applied to these trees, and both tournament and lexicase selection are supported. CDGP specifically differs from standard GP in that when evaluating a program it derives from *Spec* a query that determines if the program is correct or not. If it is not the SMT provides an input model for which the program fails, and CDGP can use this through an additional query to obtain a test pair <input, expected output> which is for evaluation in subsequent generations. We explain this in more detail using a running example of synthesizing the max 4 function. The *Spec* for max 4 is given in Figure 2.

Spec imposes constraints such that the function to be synthesized takes 4 integers as input, x,y,z, and w, and returns an integer. It further asserts that the function's output must be one of x,y,z or w and be at least greater than or equal to each of them. Any function that meets these requirements provably will provide the maximum of 4 integer inputs. From this the query to verify a synthesized program is derived. An example of this query's syntax is given in figure 3. The query defines the synthesized function and asserts that there exists an input such that the synthesized program does *not* meet the constraints of *Spec*. If no such input exists the solver returns UNSAT, meaning the program is provably correct with respect to *Spec*, and CDGP can terminate after the generation. If such an input exists, the solver returns SAT and provides a counterexample in the form of an input model I. The model I is then used for an additional query which defines the inputs to be I and assert that the input must be satisfiable with respect to the constraints of max4. Upon receipt of SAT from this call the expected output O is obtained. A test is then constructed consisting of $<I, O>$. [1]

Using the procedure above, CDGP is able to discover a training set through interaction with the SMT Solver that facilitates inductive synthesis. Note that, unlike in most instances of GP, there is no training-test set split and generalization is consequently not a concern. CDGP fits to the discovered tests and concludes either with a timeout or a program provably correct with respect to *Spec*.

2.3 Synthesis through Unification (STUN)

[2] in 2015 made the observation that smaller functions are typically easier to synthesize then larger ones. To exploit this observation [2] proposed Synthesis through Unification (STUN), which uses a combination of CEGIS and a widening operator on predicates to discover and split the input space into sub spaces where solutions could be quickly enumerated. These solutions could then be trivially unified in any grammar containing an if-than-else concept (e.g. CLIA). [4] refined this research and proposed the EUSolver, which uses a combination of enumeration and decision tree learning to determine terms and predicates which correctly map to these terms. This solver won the 2017 SyGus Competition [3] and continues to be one of the state-of-the-art solvers in the CLIA domain along with CVC4.

The original STUN and EUSolver took CEGIS concepts and achieved significant speed up through a divide and conquer methodology. We seek to achieve the same effect on CDGP with Synthesis through Unification Genetic Programming (STUN GP). Note that our method differs significantly from the original STUN algorithm, but conceptually still exploits the same properties of the CLIA domain.

[1] This interaction with the solver is described in more detail in [7].

Algorithm 1: Discovery Phase

```
Result: Partials, a list of programs covering the input
        space
Partials ← ∅;
PartialsIncorrect ← true;
while PartialsIncorrect do
    BestProgram ← CDGP(Partials);
    Partials ← Partials ∪ BestProgram;
    PartialsIncorrect ← VERIFY(Partials);
end
```

3 SYNTHESIS THROUGH UNIFICATION GENETIC PROGRAMMING (STUN GP)

Synthesis through Unification Genetic Programming (STUN GP) splits a given SyGus problem on the CLIA domain into three sequential phases: discovery, decision and unification. The first phase, discovery, uses CDGP to construct a set of programs using grammar G that cover the input space. This set, $Partials$, can under some unknown unification constraints $Unif$ be unified into a program $Correct$ that adheres to formal specification *Spec*. The second phase, decision, then finds $Unif$ by using CDGP to construct a list of predicates from G corresponding to $Partials$. These predicates map inputs to corresponding program(s) in $Partials$ such that each predicate holds strictly where the program is correct. The third phase then constructs $Correct$ by unifying $Partials$ with $Unif$ using a fast and sound unification procedure.

We describe these phases in detail below.

3.1 First Phase of STUN GP: Discovery

We say that a set of programs covers the input space on a *Spec* if for any input there is at least one program in the set that is correct. The discovery phase searches for such a set using Algorithm 1. $Partials$ is initialized as an empty list. We then run CDGP and upon termination add the best program to the list. If $Partials$ consequently covers the space, we terminate and return $Partials$, otherwise we run CDGP again. Note that we pass $Partials$ into CDGP as it is needed for internal verification calls.

3.1.1 Verification for Discovery Phase. The key modification to CDGP for the Discovery phase is that the verification function $VERIFY$ now checks for correctness on a set of programs rather than a single program. To demonstrate this we return to our example of synthesizing the function max4.

```
(set-logic LIA)
(set-option :produce-models true)
;;Now defining multiple functions

;;This defines the
;;recently synthesized program.
(define-fun max4
((x Int)(y Int)(z Int)(w Int))
Int (ite (> x w) x w))

;;This defines a program
;;found previously that
;;is partially correct.
(define-fun max4_0
((x Int)(y Int)(z Int)(w Int))
Int y)

(declare-fun x () Int) (declare-fun y () Int)
(declare-fun z () Int) (declare-fun w () Int)

;;Assert our most recent program does NOT
;;meet the specifications of Max 4.
(assert (not (and (>= (max4 x y z w) x)
(>= (max4 x y z w) y)(>= (max4 x y z w) z)
(>= (max4 x y z w) w)(or (= x (max4 x y z w))
(or (= y (max4 x y z w))(or (= z (max4 x y z w))
(= w (max4 x y z w)))))))

;;Assert our previously synthesized program
;;does NOT meet specifications of Max 4.
(assert (not (and (>= (max4_0 x y z w) x)
(>= (max4_0 x y z w) y)(>= (max4_0 x y z w) z)
(>= (max4_0 x y z w) w)(or (= x (max4_0 x y z w))
(or (= y (max4_0 x y z w))
(or (= z (max4_0 x y z w))
(= w (max4_0 x y z w)))))))

;;If there exists input on which both programs are
;;incorrect returns SAT and provides
;;counterexample.
;;Otherwise returns UNSAT meaning at least one
;;of the programs is correct for each input.
(check-sat)
(get-value (x y z w))
```

Figure 4: A verification query that determines if a set of programs is correct.

```
(set-logic LIA)
(set-option :produce-models true)
;;Program from Partials.
(define-fun max4 (
(x Int)(y Int)(z Int)(w Int)) Int w)

(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(declare-fun w () Int)

;;Our assertion.
(assert(
;;Predicate
ite(<= y w)
;;LHS, for given input produces
;;counterexample if max4 incorrect WRT input.
(not (and (>= (max4 x y z w) x)
(>= (max4 x y z w) y)
(>= (max4 x y z w) z)
(>= (max4 x y z w) w)
(or (= x (max4 x y z w))
(or (= y (max4 x y z w)) (or (= z (max4 x y z w))
(= w (max4 x y z w)))))))

;;RHS, for given input produces
;;counterexample if max4 is correct WTR input.
(and (>= (max4 x y z w) x)
(>= (max4 x y z w) y)
(>= (max4 x y z w) z)
(>= (max4 x y z w) w)
(or (= x (max4 x y z w))
(or (= y (max4 x y z w)) (or (= z (max4 x y z w))
(= w (max4 x y z w))))))
))

;;If assertion holds on any input, returns SAT
;;and provides a counterexample where input is
;;misassigned to LHS or RHS.
;;Otherwise, returns UNSAT meaning predicate
;;strictly maps inputs to LHS when max4 is
;;correct.
(check-sat)
(get-value (x y z w))
```

Figure 5: A verification query that determines if a predicate is correct with respect to the defined function.

The specification *Spec* for max4 is given in Figure 2. On the first iteration of the Discovery phase loop, $VERIFY$ is identical to the verification query in the original CDGP. On subsequent iterations, however, the query defines an additional function for each program P in $Partials$ and asserts that there exists an input such that P does not meet the constraints of max4. The syntax of this call is given in Figure 4. This query asserts that there exists an input such that *every* program defined in query does not adhere to *Spec*. If such an input exists the solver returns SAT and we receive a counterexample. If no such input exists the solver returns UNSAT and we know that the union of $Partials$ and the most recent synthesized function covers the input space.

3.1.2 Redundancy: Removing superfluous programs. A frequent outcome of Algorithm 1 is that $Partials$ contains

Algorithm 2: Decision Phase

```
Result: Unif, a list of predicates corresponding to
        Partials
Unif ← ∅;
i ← 1 //skip first program ;
while i < |Partials| do
    PredIncorrect ← true;
    while PredIncorrect do
        BestPred ← CDGP(Partials[i]);
        if VERIFYPRED(BestPred) then
            Unif ← VERIFYPRED(Partials[i],
             BestPred);
            PredIncorrect ← false;
        end
    end
    i ← i+1;
end
```

multiple programs that cover redundant areas of the input space. An example of this for the max4 problem would be a program in $Partials$ that returns x when x is greater than y, and another program that simply returns x in all cases. To account for this, we run a pruning procedure at the conclusion of Algorithm 1, iterating through $Partials$ checking each program P to see if $VERIFY(Partials \setminus P)$ returns true. If it does return true P is superfluous and is removed.[2]

3.2 Second Phase of STUN GP: Decision

The Decision phase takes the list of programs $Partials$ and computes unification constraints $Unif$ which can be used to unify $Partials$ into a correct program. $Unif$ is composed of a set of predicates, each of which corresponds to a program in $Partials$. In the general case, for any given input a correct predicate must hold if the program is correct on the input and not hold otherwise. Finding a correct predicate can thus be framed as a decision problem.

3.2.1 Verification for Decision Phase. Determining correctness of a predicate requires a new verification query, $VERIFYPRED$, and this query differs slightly depending upon problem properties. The most general case for verification is where for each possible input for a program in $Partials$ the program is correct or incorrect. In this case, finding predicate correctness can be achieved with a verification query that asserts that when the predicate holds for a given input the program is *incorrect* on this input and when the predicate does not hold the program is *correct* on this input. If this assertion holds the SMT solver returns SAT and we can obtain a counterexample. Otherwise the solver returns UNSAT, meaning that no example could be found where the predicate mapped an input to a program for which it was not correct. An example of the syntax for this call is given in Figure 5.

[2]This procedure does not guarantee a minimal set, however, in practice this is not a significant issue as larger sets of small terms are preferred.

```
(set-logic LIA)

;;Dictates the structure of the function to be
;;synthesized.
(synth-fun findIdx (
(x1 Int) (x2 Int) (k Int)) Int )

;;Declare variables in the problem statement;
(declare-var x1 Int)
(declare-var x2 Int)
(declare-var k Int)

;;Constraints dictating which index in the
;;array should be chosen, note that there
;;is an implication that x1 be strictly less than
;;x2 i.e. list must be sorted from x1 to xn.
(constraint (=> (< x1 x2) (=> (< k x1)
(= (findIdx x1 x2 k) 0))))
(constraint (=> (< x1 x2) (=> (> k x2)
(= (findIdx x1 x2 k) 2))))
(constraint (=> (< x1 x2)
(=> (and (> k x1) (< k x2))
(= (findIdx x1 x2 k) 1))))

(check-synth)
```

Figure 6: A specification for the array search 2 problem.

There are some cases for which there exists a single answer output for each input *or* the input is correct for all possible outputs. This property occurs often in problems with implied preconditions. It holds for example on the array_search_n problem where the array to be searched is implied to be sorted. In these cases, we can soundly modify the call to assert that when the predicate holds the program is incorrect on a given input and when the predicate does not hold every other program in $Partials$ is incorrect on a given input. The advantage of this is the synthesizer is directed towards the case where the implications holds rather than wasting time on the trivial case where the implication does not hold and any answer is correct. Together, these two verification queries are sound with respect to all CLIA SyGus benchmarks proposed for which CDGP is also sound.

3.2.2 Generating tests for decision problems. For CDGP to evolve solutions towards a predicate input-output examples must no longer consist of the input and expected program output. Rather they must consist of the input and whether the program corresponding to the predicate is correct for the given input. Thus, on receipt of a counterexample from $VERIFYPRED$, the input of this counterexample is evaluated on the program. If the program's output is the same as the expected output we add test pair <input, true>, otherwise we return test pair <input, false>. We henceforth call the first of these types of tests a positive and the other a negative.

Table 1: Parameters of evolutionary algorithm

Parameter	Value
Number of Runs	10
Population Size	500
Maximum height of initial programs	5
Maximum height of trees inserted by mutation	5
Maximum number of generations	2 (Discovery) /20 (Decision)
Maximum run time in seconds	3600
Probability of mutation	0.5
Probability of crossover	0.5
Selection method	Lexicase
Evolution method	Generational

We can partially control which type of test to look for by altering $VERIFYPRED$. Consider the syntax in Figure 5. If we set the LHS to be false then only positive tests will be found. Similarly, if we set the RHS to be false then only negative counterexamples will be found. By separating our verification query into these two calls, one with RHS false and the other with LHS false, we can choose to attempt to add a positive or negative test if both calls return SAT. Note that if both calls return UNSAT this is equivalent to the full call returning UNSAT and thus this can be used as a termination condition.

3.2.3 Algorithm. The algorithm for computing the predicates in $Unif$ is given by Algorithm 2. The algorithm iterates through the list $Partials$ starting from the second element and computes corresponding predicates for each. Note that CDGP uses $Partialsi$ for the internal $VERIFYPRED$ call, which checks if the predicate correctly maps inputs to and away from $Partialsi$. The termination of a successful run occurs when $Unif$ computes a sufficient number of predicates to unify $Partials$ into correct.

3.3 Final Phase of STUN GP: Unification

Our Unification phase is a simple procedure that is linear time with respect to the size of $Partials$. The procedure simply constructs $Correct$ by iterating through $Unif$ and $Partials$ mapping each predicate in $Unif$ to its corresponding program in $Partials$. This mapping is achieved through the if-than-else construct ite in our grammar G and the unmapped $Partials$ is left as the final else. This program takes the following form where $Partials$ and $Unif$ have been constructed by Algorithms 1 and 2 respectively, $Partials$ is of size n, $Unif$ is size n-1, and both are 0-indexed:

```
ite(Unif[0] Partials[1]
ite(Unif[1]  Partials[2]
ite(...
ite(Unif[n-2] Partials[n-1] Partials[0])
)))
```

This procedure is sound given the construction of $Partials$ and $Unif$ is sound as $Partials$ covers the input space, $Unif$ maps the inputs for each corresponding program in $Partials$ to strictly cover inputs for which they are correct, and the final else must be the input space covered by the unmapped $Partials$ program. Following unification we now have a full program $Correct$ constructed from grammar G whose adherence to $Spec$ we can check directly using the standard query for a single program. We perform this verification call for posterity at the end of a run.

4 EXPERIMENTS

To test the effectiveness of STUN GP we test it against a pair of SyGus benchmark problems from the SyGus competition. [3] Each problem has multiple versions with different arities, ranging from an arity of 2 to an arity of 15, giving us in total 28 benchmarks. We directly compare this to CDGP's top performing variant. We further discuss STUN GP's performance within the context of the most recently published SyGus competition results and examine the possible impact of a parallel implementation.

4.1 Configuration

Our program uses CDGP for both the Discovery phase and the Decision phase, with the only difference in configuration being the number of generations. The configuration is given in Table 1.

The generation sizes for the Discovery phase is set to only 2 per CDGP. In practice this means a random population is initialized, tests are generated through verification on this population, and the best candidate in the next generation is added to $Partials$. This does not allow the population much time to evolve, but proved well suited to the benchmarks at hand. The implications of this are discussed in the subsequent section. The Decision phase's size of 20 is still significantly shorter than the original CDGP algorithm allowed, but we found that trying shorter runs and restarting if the population does not converge proved more effective than giving poor performing initial populations many generations to converge.

Note that there is no maximum number of generations for STUN GP overall and no timeout in any of the phases. Rather, there is a global timeout of 3600 seconds in wall clock time. The original CDGP's configuration is identical to the one given by [7] except that we allow max generations

Figure 7: Median time to synthesize on benchmarks. Entries >= 3600 represent time outs.

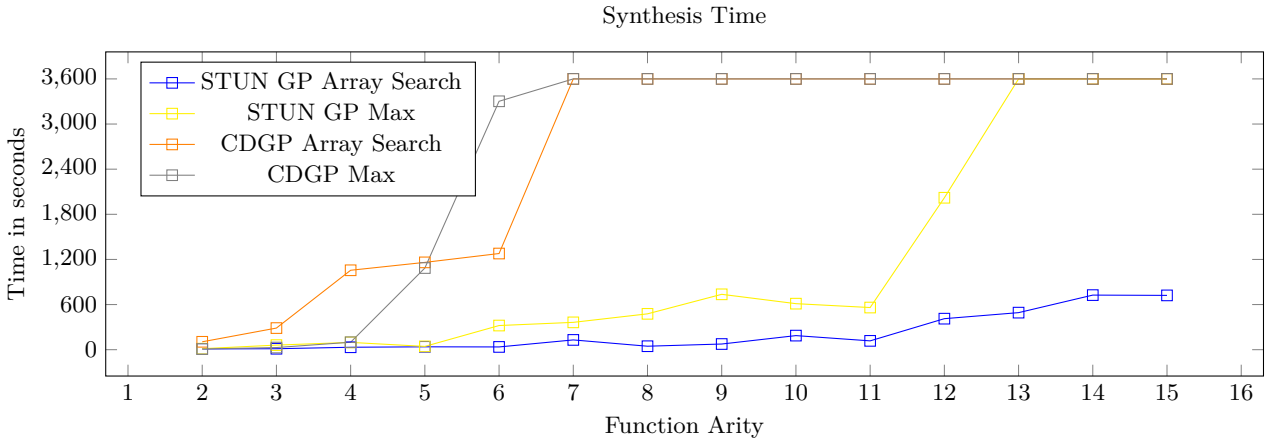
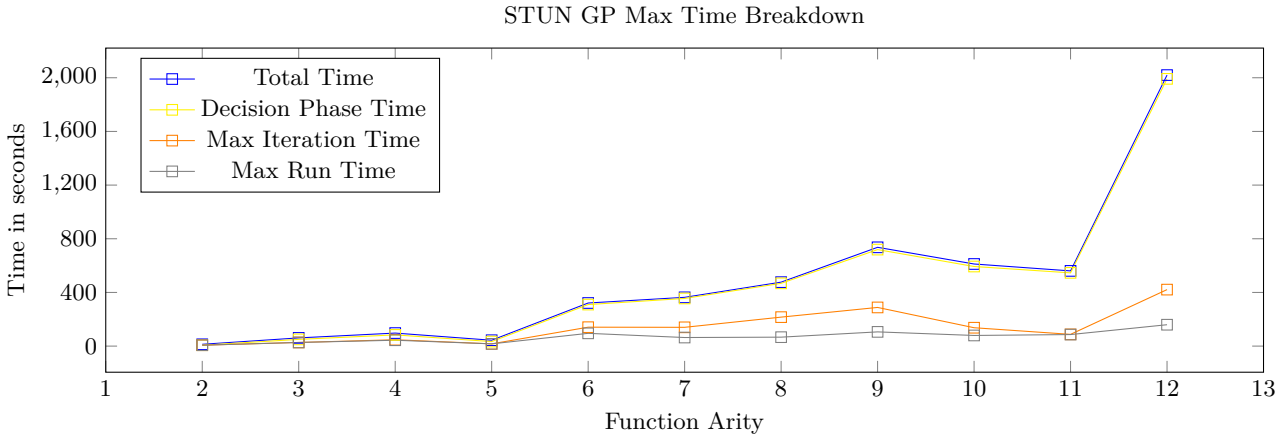


Figure 8: Time Breakdown for max problem up to function arity 12 for STUN GP.



to be up to 1000. In practice, this results in the algorithm either converging on a perfect solution or timing out. The SMT solver we interact with is Z3, a state-of-the-art SMT solver supported by Microsoft. [8]

4.2 Benchmarks

The primary metric of the SyGus Competition is wall clock time to synthesize and a benchmark is only considered solved if a solution is synthesized with 3600 seconds wall clock time. [3] The benchmarks we test against are the max_n and array_search_n benchmarks with n ranging from 2 to 15. The max_n problem requires synthesis of a function that computes the maximum of n integer inputs. Informally the array_search_n problem requires synthesis of a function that computes the location where an element k can be inserted into a sorted list of size n such that the list remains sorted. The list is composed of the first n elements passed into the function and if they are not sorted all possible outputs are true. The arity of the max_n problem is n and the arity of array_search_n is n+1. An example spec of array_search_n where n is 2 is given in Figure 6.

The SyGus competition features different versions of the max_n and array_search_n problems in their General Track and their CLIA Track. The versions in the CLIA track's only preconditions on the structure of the function to be synthesized are the number and type of inputs and the type of return value. The versions in the General Track, on the other, impose additional requirements on the structure that can help guide synthesizers towards the syntax for a given solution. The version in the CLIA track is more difficult, as the syntactic search space is larger, and consequently this is the version we test against. It is worth noting that while these problems are trivial for a human programmer to solve synthesis for higher arities has proven challenging. These problems were not both solved up to an arity of 15 until the 2nd SyGus competition in 2015. [3]

4.3 Results

STUN GP and CDGP were run for 10 trials on each of our benchmarks on a PC with an AMD Dual-Core processor. The median of these trials for time to synthesize and size is given in Figure 7. In instances where no solution was synthesized the time is set to the max time of 3600.

From Figure 7 it can be seen that STUN GP is significantly faster than CDGP as arity increases. STUN GP proved to be capable of solving 25 of the 28 benchmarks, solving all of the array_search_n benchmarks and solving up to arity 12 on the max_n problems. In contrast, CDGP only solves 8 of the 28 benchmarks, with the time to solve exploding for both problems as arity increases. The three failed benchmarks for STUN GP are a slight disappointment, but this is partially tied to increasing verification time overwhelming the run time rather than the evolutionary method failing to converge. This issue could be ameliorated with the parallel speedup discussed below especially considering that all SyGus synthesizers that use CEGIS face this bottleneck.

5 DISCUSSION

5.1 Parallel Potential

There is significant potential for parallelism with regards to the Decision phase. Each program computed in *Partials* is independent of the other and computing these concurrently would lead to a theoretical linear time speed up with respect to the number of programs in *Partials*. To demonstrate the significance of this we present Figure 8 which demonstrates on the max_n that the Decision Phase dominates the total time to solve. Consequently, while a gap still exists in performance between state-of-the-art SyGus solvers [3] a parallel implementation could further close this.

5.2 Discovery Phase

The problem of discovering terms covered by the Discovery Phase was trivial for the benchmarks tested and took little time as demonstrated in Figure 8 . Specifically, the correct terms for each benchmark in any situation can be expressed with an atomic term (e.g. x or 1) which limited the usefulness of finding more complex terms over several generations. Going forward, using enumerative methods for Discovery could be a simpler approach. It would also be worth exploring a modification of the Discovery phase that runs for more generations and then parses multiple terms and predicates out of the best solution. Multiple terms could then be added to the query at once and the predicates associated to these terms could be used to seed the populations in the Decision phase.

6 CONCLUSION

Synthesis through Unification Genetic Programming (STUN GP) represents a significant step forward towards applying GP to the SyGus problem. Our immediate next step is to produce a parallel implementation to see if the theoretical speed up is achievable in practice. We also intend to integrate recent research in the GP community with respect to using Novelty Search as a diversity driver.[10][9] This approach has demonstrated promising results in the general Program Synthesis domain that might hold for the SyGus problem.

STUN GP builds on the excellent work of CDGP [11] in integrating research from the Formal Methods community into GP such that provably correct programs can be synthesized. A good question is whether or not there are potential benefits from GP research that can be integrated into traditional Formal Methods approaches to Program Synthesis. GP's inherent properties lends themselves to hybridization with these methods. Such a hybridization could thus provide a well motivated way to transform state-of-the-art deterministic solvers into stochastic ones that can exploit the corpus of Genetic Programming to improve scalability. We consider this to be a promising long term research direction.

The authors were supported by EPSRC, EP/R018472/1.

REFERENCES

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis*. IEEE.

[2] Rajeev Alur, Pavol Černỳ, and Arjun Radhakrishna. 2015. Synthesis through unification. In *International Conference on Computer Aided Verification*. Springer, 163–179.

[3] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438* (2017).

[4] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 319–336.

[5] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-based program synthesis. *Commun. ACM* 61, 12 (2018), 84–93.

[6] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

[7] Iwo Błądek, Krzysztof Krawiec, and Jerry Swan. 2018. Counterexample-driven genetic programming: Heuristic program synthesis from formal specifications. *Evolutionary computation* 26, 3 (2018), 441–469.

[8] 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.

[9] Lia Jundt and Thomas Helmuth. 2019. Comparing and combining lexicase selection and novelty search. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 1047–1055.

[10] Jonathan Kelly, Erik Hemberg, and Una-May O'Reilly. 2019. Improving genetic programming with novel exploration-exploitation control. In *European Conference on Genetic Programming*. Springer, 64–80.

[11] Krzysztof Krawiec, Iwo Błądek, and Jerry Swan. 2017. Counterexample-driven genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 953–960.

[12] Michael O'Neill and Lee Spector. 2019. Automatic programming: The open issue? *Genetic Programming and Evolvable Machines* (2019), 1–12.

[13] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. 2008. *A field guide to genetic programming*. Lulu. com.