SyGuS-Comp 2016: Results and Analysis

Rajeev Alur
University of Pennsylvania

Dana Fisman
Ben-Gurion University

Rishabh Singh
Microsoft Research, Redmond

Armando Solar-Lezama
Massachusetts Institute of Technology

Syntax-Guided Synthesis (SyGuS) is the computational problem of finding an implementation f that meets both a semantic constraint given by a logical formula φ in a background theory T, and a syntactic constraint given by a grammar G, which specifies the allowed set of candidate implementations. Such a synthesis problem can be formally defined in SyGuS-IF, a language that is built on top of SMT-LIB.

The *Syntax-Guided Synthesis Competition (SyGuS-Comp)* is an effort to facilitate, bring together and accelerate research and development of efficient solvers for SyGuS by providing a platform for evaluating different synthesis techniques on a comprehensive set of benchmarks. In this year's competition we added a new track devoted to *programming by examples*. This track consisted of two categories, one using the theory of bit-vectors and one using the theory of strings. This paper presents and analyses the results of SyGuS-Comp'16.

1 Introduction

The Syntax-Guided Synthesis Competition (SyGuS-Comp) was originally developed as a community effort in order to provide an objective basis to compare different approaches to solving the Syntax-Guided Synthesis problem. In this style of synthesis, the user provides a specification in the form of a logical formula φ in a background theory T, and a space of programs given as a grammar G; the goal of the synthesizer is to find a program in the space that satisfies the given specification. Concretely, if the specification uses an unknown function f, the goal is to find a function f_{imp} that is expressible in the grammar G and such that the formula $\varphi[f/f_{imp}]$ is valid for all values of its free variables.

One of the achievements of the community effort behind SyGuS-Comp has been the development of a standard format for benchmarks. The SyGuS format is detailed in other publications [7, 2],[22, 5, 6], but at a high-level, the SyGuS format is based on the popular SMT-LIB format for defining SMT problems and is extended to support the description of function grammars. The SyGuS format has been extended over the last two years to provide special support for important classes of problems such as Invariant Synthesis, or problems involving expressions in integer linear arithmetic [5]. In its 2016 iteration, the format was also extended to support Programming by Example problems [6], which are becoming an important area of study in the synthesis community.

In the short time that the formalism has been in public circulation, it has already performed well in its goal of facilitating research in synthesis while providing a basis for objective comparison of different algorithms. For example, the competition has provided important insights into the relative merits of different algorithms [3, 2, 7] which have been exploited to help develop and evaluate new algorithms [11, 15, 24, 26, 18, 4, 12]. Beyond synthesizer developers, there is a growing community of users that is coalescing around the formalism.

SyGuS has found various interesting applications among which are motion planning [8], compiler optimizations, and cybersecurity [9]. Remarkably, Eldib et al. report that a circuit for mitigating time-

Dimitrova, Piskac (Eds.): Fifth Workshop on
Synthesis (SYNT 2016)
EPTCS 229, 2016, pp. 178–202, doi:10.4204/EPTCS.229.13

delay attacks generated via SyGuS is much smaller than a handcrafted circuit mitigating the same attack, as well as the original circuit (which is vulnerable to that attack). The SyGuS generated circuit used 13 gates compared to 41 of the handcrafted circuit and 21 of the original, and has a shorter critical path: 3 unit delay vs. 6 unit delay of the mitigated and original circuits.

1.1 The General Track

We now illustrate the key ideas behind the main formalism and the extensions that have been added in the last two years through a series of illustrative examples.

Example We illustrate the general SyGuS-Comp formalism with this simple example from one of the competition benchmarks. This example is taken from an implementation of a quantum control computer (QCC).[1] The QCC uses expressions from the following grammar:

$$g ::= c \mid g+g \mid g-g \mid g?g$$

where c is any integer constant, $+$ is addition, $-$ is substraction, and $a?b$ stands for "if $a \geq 0$ then a else b". This minimal set of instruction is used to enable a fast implementation. High level commands should be translated to this grammar using a minimal number of operations, since these operations participate in a pipeline, thus every unnecessary delay multiplies. The goal in the following benchmark is to find two functions `qm-inner-loop` and `qm-outer-loop` that decrement an inner and outer loop (the inner from 7 to 0, the outer from 3 to 0) formally defined as follows for $x \geq 0$ and $y \geq 0$.

$$\texttt{qm-inner-loop}(x) = \begin{cases} 7 & \text{if } x = 0 \\ x-1 & \text{if otherwise} \end{cases}$$

$$\texttt{qm-outer-loop}(x,y) = \begin{cases} 3 & \text{if } x = 0 \ \wedge \ y = 0 \\ y-1 & \text{if } x = 0 \ \wedge \ y \neq 0 \\ y & \text{otherwise} \end{cases}$$

These constraints can be succinctly expressed in the SyGuS format as shown below.

```
(set-logic LIA)

(define-fun qm ((a Int) (b Int)) Int
               (ite (< a 0) b a))

(synth-fun qm-inner-loop ((x Int)) Int
   ((Start Int (x
                0
                1
                7
               (- Start Start)
               (+ Start Start)
               (qm Start Start)))))
```

[1] We thank Nissim Ofek (Yale) for contributing these benchmarks.

```
(synth-fun qm-outer-loop ((x Int) (y Int)) Int
   ((Start Int (x
                y
                0
                1
                3
                (- Start Start)
                (+ Start Start)
                (qm Start Start)))))



(declare-var x Int)
(declare-var y Int)

(constraint (or (< x 0))
            (= (qm-inner-loop x)
               (ite (= x 0) 7 (- x 1))))


(constraint (or (or (< x 0) (< y 0))
            (= (qm-outer-loop x y)
               (ite (= x 0) (ite (= y 0) 3 (- y 1))
                            y))))

(check-synth)
```

The `define-fun` command provides the description of the '?' or `qm` primitive function:

$$\texttt{qm}(a,b) = \begin{cases} a & \text{if } a \geq 0 \\ b & \text{otherwise} \end{cases}$$

The `set-logic` directive indicates that the constraints should be interpreted in terms of the theory of linear integer arithmetic. The directive `declare-var` is used to declare `x` and `y` as universally quantified integer variables. The constraints are introduced with the directive `constraint`, and `check-synth` marks the end of the problem and prompts the synthesizer to solve for the missing function. Crucially, in order for the synthesizer to generate `qm-inner-loop` and `qm-outer-loop`, it needs a grammar, which is provided as part of the `synth-fun` directive. The specified grammar provides exactly the set of allowed operations for the QCC.

1.2 Conditional Linear Integer Arithmetic Track

For problems where the grammar consists of the set of all possible integer linear arithmetic terms, it is sometimes possible to apply specialized solution techniques that exploit the information that decision procedures for integer linear arithmetic are able to produce. The 2015 SyGuS competition included a separate track where the grammar for all the unknown functions was assumed to be the entire theory of Integer Linear Arithmetic with ITE conditionals.

Example As a simple example, consider the problem of synthesizing a function `abs` that produces the absolute value of an integer. The problem can be specified with the constraint below:

```
(set-logic LIA)
(synth-fun abs ((x Int)) Int)
(declare-var x Int)
(constraint (>= (abs x) 0))
(constraint (or (= x (abs x)) (or (= (- x) (abs x)))))
(check-synth)
```

Note that the definition of the unknown function `abs` does not include a grammar this time, but because the problem is defined in the theory of linear integer arithmetic (`LIA`), the default grammar consists of all the operations available in the theory.

1.3 Invariant Synthesis Track

One of the main applications of SyGuS is invariant synthesis. For this problem, the goal is to discover an invariant that makes the verification condition for a given loop valid. Such a problem can be easily encoded in SyGuS, but invariant synthesis problems have structure that some solution algorithms are able to exploit and that can be lost when encoding it into SyGuS. Like the 2015 competition, the 2016 competition also included a separate track for invariant synthesis problems where the additional structure is made apparent. In the invariant synthesis version of the SyGuS format, the constraints are separated into pre-condition, post-condition and transition relation, and the grammar for the unknown invariant is assumed to be the same as that for the conditional linear arithmetic track. We illustrate this format with an example from last year's report [7].

Example For example, consider the following simple loop.

```
Pre: i >= 0  and  j=j0 and i=i0;
while(i > 0){
    i = i - 1;
    j = j + 1;
}
Post: j = j0 + i0;
```

Suppose we want to prove that the value of j at the end of the loop equals the value of i + j at the beginning of the loop. The verification condition for this loop would check that (a) the precondition implies the invariant, (b) that the invariant is inductive, so if it holds before an iteration and the loop condition is true, then it will hold after that iteration, and (c) that the invariant together with the negation of the loop condition implies the postcondition. All of these constraints can be expressed in the standard SyGuS format, but they can be expressed more concisely using the extensions explicitly defined for this purpose. Specifically, the encoding will be as follows.

```
(set-logic LIA)

(synth-inv inv-f ((i Int) (j Int) (i0 Int) (j0 Int)))

(declare-primed-var i0 Int)
(declare-primed-var j0 Int)
```

```
(declare-primed-var i  Int)
(declare-primed-var j  Int)

(define-fun pre-f ((i Int) (j Int) (i0 Int) (j0 Int)) Bool
                  (and (>= i 0) (and (= i i0) (= j j0))))

(define-fun trans-f ((i Int) (j Int) (i0 Int) (j0 Int)
                     (i! Int) (j! Int) (i0! Int) (j0! Int)) Bool
                     (and (and (= i! (- i 1)) (= j! (+ j 1)))
                          (and (= i0! i0) (= j0! j0))))

(define-fun post-f ((i Int) (j Int) (i0 Int) (j0 Int)) Bool
                   (= j (+ j0 i0)))

(inv-constraint inv-f pre-f trans-f post-f)

(check-synth)
```

The directive `(declare-primed-var i)` is equivalent to separately declaring `i` and `i!`, where the primed version of the variables is used to distinguish their value before and after the loop body. Just like in the earlier example, the function to be synthesized `inv_f` does not include a grammar, so the entire `LIA` grammar is assumed. The constraint `inv-constraint` is syntactic sugar for the full verification condition involving the invariant, precondition, postcondition and transition function.

1.4 Programming By Example Track

There has been a lot of recent interest in the synthesis community for learning programs from examples. Programming By Examples (PBE) systems have been developed for many domains including string transformations [13, 14, 28], data structure manipulations [29, 30], interactive parser synthesis [17], higher-order functional programs over recursive data types [21, 10], and program refactorings [23]. The 2016 competition included a new separate track for Programming by Examples. The grammar for benchmarks in this track is specified using a context-free grammar similar to the general SyGuS track, but the specification constraints can only be specified using input-output examples. The benchmarks in this track included theory of integers, bit-vectors, and strings.

Example Consider the following task taken from FlashFill [13, 14] that requires learning a string transformation program that constructs the initials of the first and last names.

Input	Output
Nancy FreeHafer	N.F.
Andrew Cencici	A.C.
Jan Kotas	J.K.
Mariya Sergienko	M.S.

Table 1: a FlashFill example task

The encoding of this problem in the PBE track is as follows:

```
(set-logic SLIA)

(synth-fun f ((name String)) String
    ((Start String (ntString))
     (ntString String (name " " "."
                       (str.++ ntString ntString)
                       (str.replace ntString ntString ntString)
                       (str.at ntString ntInt)
                       (int.to.str ntInt)
                       (str.substr ntString ntInt ntInt)))
      (ntInt Int (0 1 2
                  (+ ntInt ntInt)
                  (- ntInt ntInt)
                  (str.len ntString)
                  (str.to.int ntString)
                  (str.indexof ntString ntString ntInt)))
      (ntBool Bool (true false
                    (str.prefixof ntString ntString)
                    (str.suffixof ntString ntString)
                    (str.contains ntString ntString)))))

(declare-var name String)

(constraint (= (f "Nancy FreeHafer") "N.F."))
(constraint (= (f "Andrew Cencici") "A.C."))
(constraint (= (f "Jan Kotas") "J.K."))
(constraint (= (f "Mariya Sergienko") "M.S."))

(check-synth)
```

The benchmark uses SMT-LIB's SLIA theory that encodes several string functions such as `str.len`, `str.indexof`, `str.contains` etc. All the constant strings that are needed to perform the transformation are also provided as part of the grammar.

1.5 SyGuS-Comp'14 summary

The first SyGuS competition, SyGuS-Comp'14 consisted of a single track — the general track — in which the benchmark provides the grammar describing the desired syntactic restrictions for that benchmark. The background theory could be either linear interger arithmetic or bitvectors. Five solvers competed in SyGuS-Comp'14. The solver who won the first place was the ENUMERATIVE solver which solved 126 out of 241 benchmarks.

1.6 SyGuS-Comp'15 summary

The 2015 instance of SyGuS-Comp was the second iteration of the competition and the first iteration to include the separate conditional linear integer arithmetic and invariant synthesis tracks. There were a

total of eight solvers submitted to the competition which represented a range of solution strategies. The CVC4-1.5 solver won the general track and the conditional linear integer arithmetic tracks, whereas the ICE DT solver won the invariant synthesis track.

1.7 SyGuS-Comp'16 summary

The 2016 instance of SyGuS-Comp was the third iteration of the competition and included an additional track on Programming By Examples (PBE). In addition to the previous solvers, there were two new solver submitted this year: CVC4-1.5.1 and EUSolver. In the rest of the paper, we describe the details of the benchmarks, new solver strategies, and the results of the competition on different benchmark categories.

2 Competition Settings

2.1 Participating Benchmarks

In addition to last year's competition benchmarks, we had 858 new benchmarks for the Programming By Example (PBE) track. For other tracks, we had the same benchmarks as of last year: General Track (309), CLIA Track (73), and Invariant Synthesis Track (67).

The benchmarks in the PBE track can be classified into two categories:

- **String Transformations:** The 108 string transformation tasks are taken from public benchmarks of FlashFill [13, 14] and BlinkFill [28]. The transformations are defined using a Domain-specific language of string transformations that involve concatenation of substrings of input strings and constant strings, where the substring expressions involve learning positions corresponding to $k^{\texttt{th}}$ occurrence of a constant string in the inputs.
- **Bitvector Transformations:** The 450 bitvector transformation benchmarks were obtained from the 2013 ICFP Programming Competition[2] [1]. The programs for the benchmarks were sampled from a bitvector DSL using a strategy of construction k-nuggets (programs of size k that are minimal) and then composing them to generate larger programs. An additional 300 bitvector benchmarks using the same grammar were submitted by Arjun Radhakrishna.

String Benchmarks The string benchmarks were taken from the public string transformation benchmarks in FlashFill and BlinkFill. These benchmarks correspond to common data cleaning tasks faced by spreadsheet users. The hypothesis space of possible transformations is defined by a DSL that is expressive enough to encode majority of common tasks but at the same time amenable for efficient learning. A subset of the DSL that was encoded in SyGuS benchmark is shown in Figure 1. Note that the SyGuS grammar for these benchmarks currently does not contain loops (Kleene star) and regular expression based position expressions.

The grammar at the top-level consists of string concatenation (`str.++`) expressions involving constant strings and substring expressions. The constant strings needed for each benchmark are also provided in each benchmark (`c1`, `c2`, etc.). For some of the string transformation benchmarks, we created two additional class of benchmarks with the suffix `-long` and `-repeat`. The `-long` benchmarks had 100 input-output examples, whereas the `-repeat` benchmarks consisted of several input-output examples that were repeated in the constraint. The goal of these additional benchmark categories was to see

[2] http://icfpc2013.cloudapp.net/

```
(synth-fun f ((x String) (y String)) String
    ((Start String (ntString))
     (ntString String (x y c1 c2 ...
                   (str.++ ntString ntString)
                   (str.replace ntString ntString ntString)
                   (str.at ntString ntInt)
                   (int.to.str ntInt)
                   (ite ntBool Start Start)
                   (str.substr ntString ntInt ntInt)))
      (ntInt Int (0 1 2 (+ ntInt ntInt) (- ntInt ntInt)
                   (str.len ntString) (str.to.int ntString)
                   (str.indexof ntString ntString ntInt)))
      (ntBool Bool (true false
                   (str.prefixof ntString ntString)
                   (str.suffixof ntString ntString)
                   (str.contains ntString ntString)))))
```

Figure 1: The grammar for string transformation benchmarks in the PBE track.

how increasing the number of examples affects the solver performance, and if solving algorithms can avoid reasoning about repeated input-output examples.

Bitvector Benchmarks The bitvector benchmarks were taken from the 2013 ICFP programming contest and the DSL encoded as a SyGuS grammar for the benchmarks is shown in Figure 2. Similar to the string transformation DSL, the constants needed for the desired transformation are provided in the grammar.

The benchmarks for this category were generated from the DSL by first sampling k-nuggest from the DSL and then composing them to obtain larger programs. A k-nugget is a program expression in the DSL of size k such that no other expression in the DSL of size less than k is equivalent with it. The idea in using the k-nuggets for program generation is that the composed programs would lead to more challenging programs that will be less likely to be solved by synthesizing a small equivalent program in the DSL.

2.2 Participating Solvers

In addition to 7 solvers from last year's competition, we had two new solver submissions for the 2016 competition: i) CVC4 1.5.1 and ii) EUSolver. Table 2 summarizes which solver participated in which track. The two new solvers participated in all 4 tracks. A total of 6 solvers participated in the General track, 5 in the invariant synthesis track, and 5 in the Conditional Linear arithmetic track. Figure 2 lists the submitted solvers together with their authors.

The CVC4-1.5.1 solver employs a refutation-based synthesis approach [24]. Instead of solving an exists-forall synthesis formula, it first negates the formula to obtain a forall-exists problem and tries to show it is unsatisfiable. It eliminates the forall quantification over unknown function in two ways: i) if the function is always called with the same parameters in the formula, it skolemizes it with a first-

```
(define-fun shr1 ((x (BitVec 64))) (BitVec 64) (bvlshr x #x0000000000000001))
(define-fun shr4 ((x (BitVec 64))) (BitVec 64) (bvlshr x #x0000000000000004))
(define-fun shr16 ((x (BitVec 64))) (BitVec 64) (bvlshr x #x0000000000000010))
(define-fun shl1 ((x (BitVec 64))) (BitVec 64) (bvshl x #x0000000000000001))
(define-fun if0 ((x (BitVec 64)) (y (BitVec 64)) (z (BitVec 64))) (BitVec 64)
  (ite (= x #x0000000000000001) y z))

(synth-fun f ( (x (BitVec 64))) (BitVec 64)
  ((Start (BitVec 64) (c1 c2 ...  x (bvnot Start)
       (shl1 Start) (shr1 Start)
       (shr4 Start) (shr16 Start)
       (bvand Start Start) (bvor Start Start)
       (bvxor Start Start) (bvadd Start Start)
       (if0 Start Start Start)))))
```

Figure 2: The grammar for bitvector synthesis benchmarks in the PBE track.

	Solvers								
Tracks	ALCHEMIST-CS	ALCHEMIST-CSDT	CVC4-1.5	ENUMERATIVE	ICE-DT	SKETCH-AC	STOCHASTIC	CVC4-1.5.1	EUSOLVER
LIA	1	1	1	0	0	0	0	1	1
INV	1	0	1	0	1	0	0	1	1
General	0	0	1	1	0	1	1	1	1
PBE	0	0	1	1	0	1	1	1	1

Table 2: Solvers participating in each track

order variable (single invocation case), ii) otherwise if the single invocation property does not hold in the formula, it uses a syntax-guided approach for restricting the space of functions using the grammar. This year's submission had new improvements in both of these two ways. For the single invocation case, the solver has a termination guarantee for LIA and supports newer ways to recognize when a property can be rewritten as single invocation. For the syntax-guided case, it supports an improved symmetry breaking and adds optimizations for unfolding of evaluation functions. For the invariant track, it fixes templates for unknown invariants to improve scalability of the solving algorithm.

The EUSOLVER combines enumeration with unification to learn complex functions from a grammar that satisfy the specification. It first learns small terms from the function grammar using enumeration such that the learnt terms cover the set of all points. It then synthesizes larger expressions by enumerating predicates and combining them with the learnt terms using a decision tree learning algorithm. It supports multiple sophisticated algorithms for term generation, predicate generation, and unification to compose larger expressions for different categories of benchmarks.

Solver	Authors
ALCHEMIST-CS	Daniel Neider (UIUC), Shambwaditya Saha (UIUC) and P. Madhusudan (UIUC)
ALCHEMIST-CSDT	Shambwaditya Saha (UIUC), Daniel Neider (UIUC) and P. Madhusudan (UIUC)
CVC4-1.5	Andrew Reynolds (EPFL), Viktor Kuncak (EPFL), Cesare Tinelli (Univ. of Iowa), Clark Barrett (NYU), Morgan Deters (NYU) and Tim King (Verimag)
ENUMERATIVE	Abhishek Udupa (Penn)
ICE-DT	Daniel Neider (UIUC), P. Madhusudan (UIUC) and Pranav Garg (UIUC)
SKETCH-AC	Jinseong Jeon (UMD), Xiaokang Qiu (MIT), Armando Solar-Lezama (MIT) and Jeffrey S. Foster (UMD)
STOCHASTIC	Mukund Raghothama (Penn)
CVC4-1.5.1	Andrew Reynolds (Univ. Of Iowa), Cesare Tinelli (Univ. of Iowa), Clark Barrett (NYU), and Tim King (Google)
EUSOLVER	Arjun Radhakrishna (Penn) and Abhishek Udupa (Microsoft)

Figure 3: Submitted solvers and their authors

2.3 Experimental Setup

The solvers were run on the StarExec platform [34] with a dedicated cluster of 12 nodes, where each node consisted of two 4-core 2.4GHz Intel processors with 256GB RAM and a 1TB hard drive. The memory usage limit of each solver run was set to 128GB. The wallclock time unit was set to 3600 seconds (thus, a solver that used all cores could consume at most 14400 seconds cpu time).

The solution that the solvers produce are being checked for both syntactic and semantic correctness. That is, a first post-processor checks that the produced expression adheres to the grammar specified in the given benchmark, and if this check passes, a second post-processor checks that the solution adheres to semantic constraints given in the benchmark (by invoking an SMT solver).

3 Competition Results and Analysis

3.1 Results Overview

The combined results for all tracks for each benchmark is shown in Figure 4. The figure shows the sum of percentages of benchmarks solved by the solvers for each category. We can observe that the EUSOLVER solves the highest percentage of benchmarks in the combined tracks, whereas the CVC4-1.5.1 solver solves the second most percentage of benchmarks.

The primary criterion for winning a track was the number of benchmarks solved, but we also analyzed the time to solve and the the size of the generated expressions. Both where classified using a pseudo-logarithmic scale as follows. For time to solve the scale is [0,1), [1,3), [3,10), [10,30),[30, 100), [100,300), [300, 1000), [1000,3600), $>$3600. That is the first "bucket" refers to termination in less than one second, the second to termination in one to three second and so on. We say that a solver solved a certain benchmark *among the fastest* if the time it took to solve that benchmark was on the same bucket as that of the solver who solved that benchmark the fastest. For the expression sizes the pseudo-logarithmic scale we used is [1,10), [10,30), [30,100), [100,300), [300,1000), $>$1000 where expression size is the number of nodes in the SyGuS parse-tree. We also report on the number of benchmarks *solved uniquely* by a solver (meaning the number of benchmark that solver was the single solver that managed to solve them).

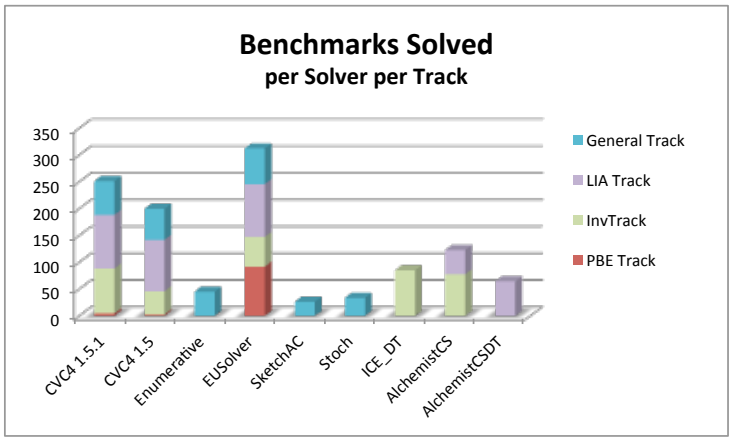

Figure 4: The overall combined results for each solver on benchmarks from all the 4 tracks.

General Track The percentage of benchmarks solved by each solver in the General track is shown in Figure 5 on the top left. The EUSOLVER solves the maximum number of benchmarks 206 out of 309. The CVC4-1.5.1 solver solves 195 benchmarks, whereas the last year's winner in this category CVC4-1.5 solved 179 benchmarks. The EUSOLVER solved 59 benchmarks uniquely and CVC4-1.5.1 solved 22 benchmarks uniquely. With regard to time to solve, the CVC4-1.5.1 solvers solved 161 benchmarks among the fastest whereas the EUSOLVER solved 127 benchmarks among the fastest. For details on the expression size see Figures 6 to 9.

Conditional Linear ArithmeticTrack The percentage of benchmarks solved by the solvers in the Conditional Linear Integer Arithmetic track is shown in Figure 5 on the top right. The CVC4-1.5.1 solver solved all 73 benchmarks in this category, whereas the EUSOLVER solved 72 out of the 73 benchmarks. Last year's winner in this category, CVC4-1.5, solved 70 benchmarks. One benchmark was solved uniquely, by CVC4-1.5.1. The CVC4-1.5.1 solver solved 72 benchmarks among the fastest and EUSOLVER solved 33 among the fastest.

Invariant Synthesis Track The result for the invariant synthesis track is shown in Figure 5 on the bottom left. In this track, the ICE-DT solver (also last year's track winner) solves the maximum number of benchmarks 57 out of 67. The CVC4-1.5.1 solver solves 56 benchmarks, whereas the ALCHEMIST-CSDT solver solves 52 benchmarks. Two benchmarks were solved uniquely, the two by ICE-DT. In terms of time to solve CVC4-1.5.1 preformed best, solving 50 bechmarks among the fastest. This is an impressive improvement from last years' version CVC4-1.5 which solved 10 benchmarks among the fastest. The ICE-DT solver solved 44 benchmarks among the fastest and the ALCHEMIST-CSDT solver solved 37 benchmarks among the fastest.

Programming By Example Track The results for the new Programming By Example (PBE) track is shown in Figure 5 on the bottom right. Unlike other tracks, we see a dramatic difference in the performance of the solvers for the benchmarks in the PBE track. The EUSOLVER remarkably solves 787 benchmarks out of 858 (742 out of 745 in the bit-vectors category and 45 out of 108 in the strings category), whereas the second best solver CVC4-1.5.1 solves 39 benchmarks (21 in the bit-vectors category and 18 in the strings category). No other solver solved more than 1 problem in this track.

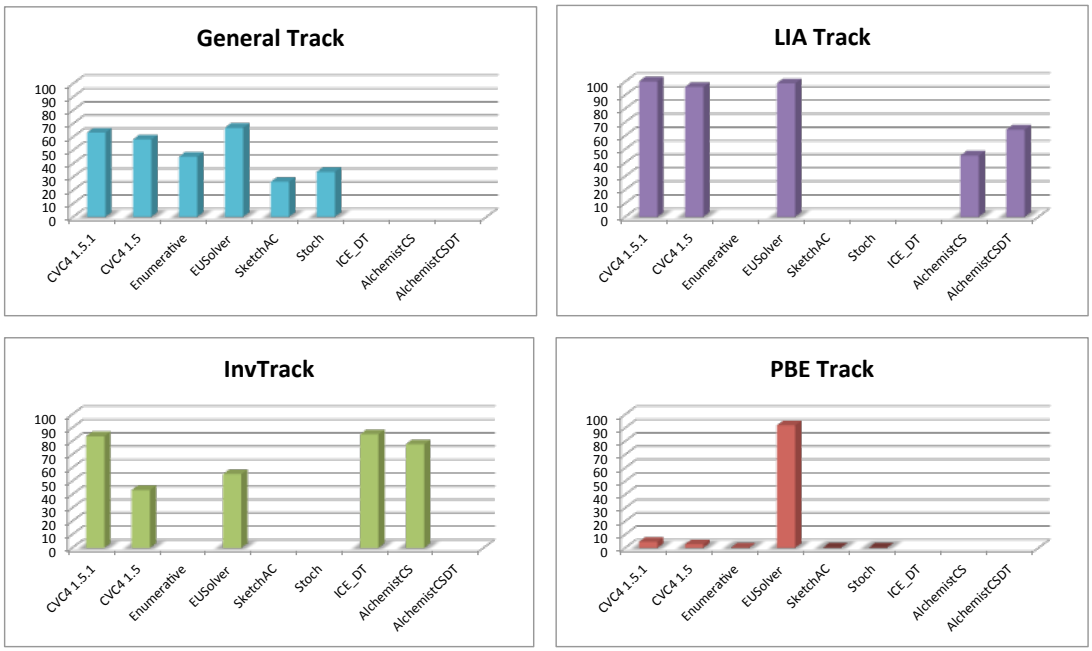

Figure 5: The percentage of benchmarks solved by the different solvers in each of the four tracks

The EUSOLVER solved 751 benchmarks uniquely (720 in the bit-vectors category and 31 in the strings category), and CVC4-1.5.1 solved 4 benchmarks uniquely (all in the strings category).

3.2 Detailed Results

In the following section we show the results of the competition from the benchmark's perspective. For a given benchmark we would like to know: how many solvers solved it, what is the min and max time to solve, what are the min and max size of the expressions produced, which solver solved the benchmark the fastest, and which solver produced the smallest expression.

We represents the results per benchmarks in groups organized per tracks and categories. For instance, Fig. 6 at the top presents details of the compiler optimization benchmarks. The black bars show the range of the time to solve among the different solvers in pseudo logarithmic scale (as indicated on the upper part of the y-axis). Inspect for instance benchmark `qm_choose_01.sl`. The black bar indicates that the fastest solver to solve it used less than 1 second, and the slowest used between 100 to 300 seconds. The black number above the black bar indicates the exact number of seconds (floor-rounded to the nearest second) it took the slowest solver to solve a benchmark (and ∞ if at least one solver exceeded the time bound). Thus, we can see that the slowest solver to solve `qm_choose_01.sl` took 199 seconds to solve it. The white number at the lower part of the bar indicates the time of the fastest solver to solve that benchmark. Thus, we can see that the fastest solver to solve `qm_choose_01.sl` required less than 1 second to do so. The colored squares/rectangles next to the lower part of the black bar, indicate which solvers were the fastest to solve that benchmark (according to the solvers' legend at the top). Here, *fastest* means in the same logarithmic scale as the absolute fastest solver. For instance, we can see that ENUMERATIVE, STOCHASTIC, and EUSOLVER were the fastest to solve `qm_choose_01.sl`, solving it in less than a second and that among the 2 solvers that solved `MPwoL_d5s3.sl` only ENUMERATIVE

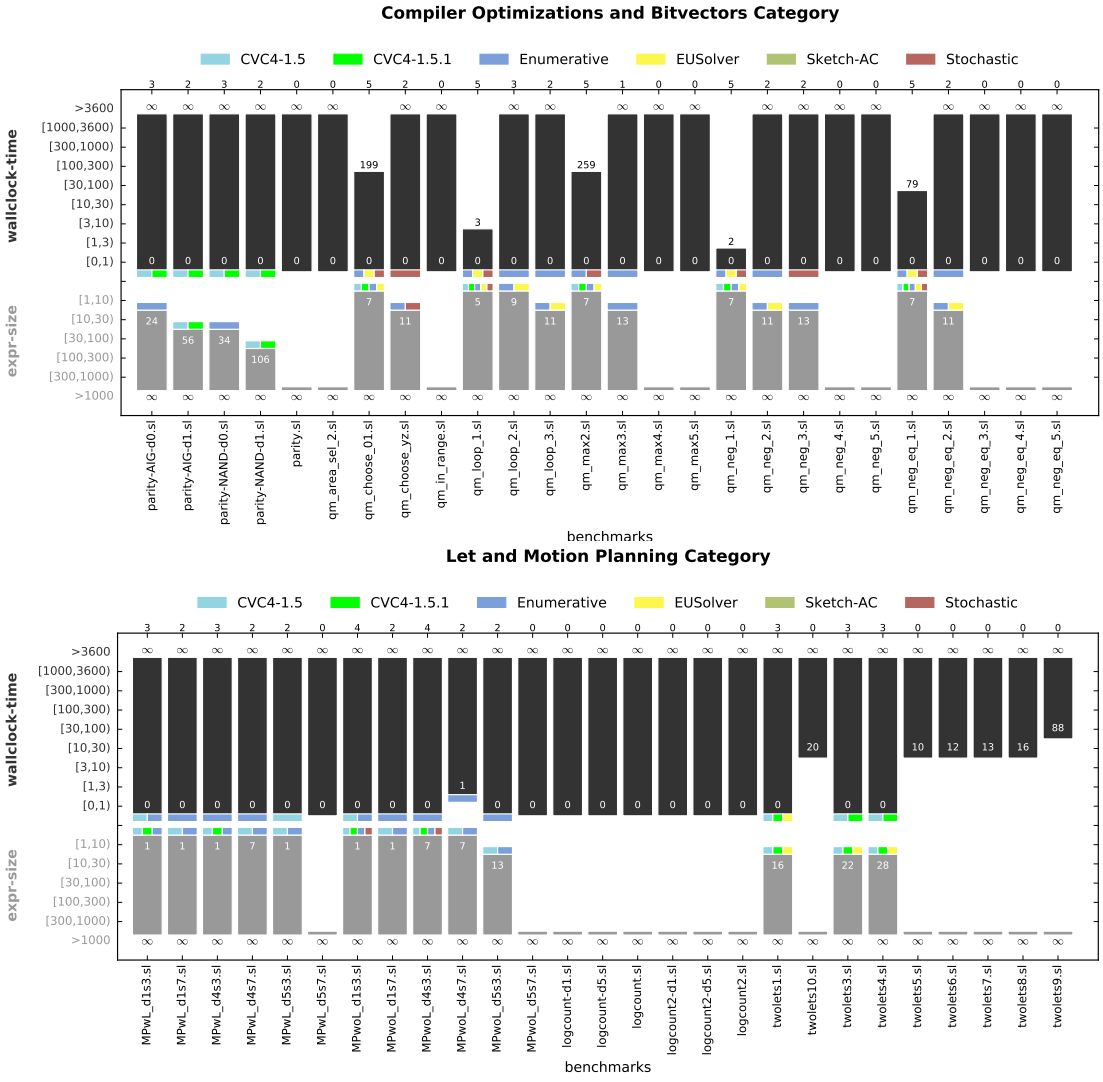

Figure 6: Evaluation of Compiler Optimizations, BitVectors, Lets and Motion Planning benchmarks.

solved it in less than 3 seconds.

Similarly, the gray bars indicate the range of expression sizes in pseudo logarithmic scales (as indicated on the lower part of the y-axis), where the size of an expression is determined by the number of nodes in its parse tree. The black number at the bottom of the gray bar indicates the exact size expression of the largest solution (or ∞ if it exceeded 1000), and the white number at the top of the gray bar indicates the exact size expression of the smallest solution. The colored squares/rectangles next to the upper part of the gray bar indicates which solvers (according to the legend) produced the smallest expression (where *smallest* means in the same logarithmic scale as the absolute smallest expression). For instance, for `qm_choose_01.sl` the smallest expression produced had size 7, and 4 solvers out of the 5 who solved it managed to produce an expression of size less than 10.

Finally, at the top of the figure above each benchmark there is a number indicating the number of solvers that solved that benchmark. For instance, one solver solved `qm_max3.sl`, two solvers solved `qm_neg2.sl`, three solvers solved `qm_loop2.sl`, and no solver solved `qm_max4.sl` or `twolets10.sl`. Note that the reason `twolest10.sl` has 20 as the lower time bound, is that that is the time to terminate rather than the time to solve. Thus, one of the solvers has terminated within 20 seconds, but either it did not produce a result, or it produced an incorrect result. When no solver produced a correct result, there will be no colored squares/rectangles next to the lower parts of the bars.

3.3 Observations

Analyzing the results of the general track per category (see Figure 13), along the number of benchmarks solved, the number of benchmarks solved uniquely and the number of benchmarks solved among the fastest, we can see that each category of the general track has a clear winner:

- CVC4-1.5.1 won 4 categories: Arrays, Let & Motion Planning, Hackers' Delight and Integers.
- ENUMERATIVE won 3 categories: Compiler-Optimizations, Invariant Generation and Invariant Generation with unbounded integers.
- EUSOLVER won 3 categories: Multiple Functions and ICFP.

If we disregard the partition to categories we can make the following observations:

- EUSOLVER solved more benchmarks of the general track than all other solvers
  - EUSOLVER solved 206/309,
  - CVC4-1.5.1 solved 195/306, and
  - ENUMERATIVE solved 139/309.
- In terms of time to solve, CVC4-1.5.1 solved more benchmarks among the fastest
  - CVC4-1.5.1 solved 157 among fastest,
  - EUSOLVER solved 123 among fastest, and
  - ENUMERATIVE solved 114 among fastest.

With regard to expression sizes, we see that in average CVC4-1.5.1 and EUSOLVER generate large expressions. The average expression size for CVC4-1.5.1 is 31580.5 and for EUSOLVER it is 30595.7 whereas the average sizes of ENUMERATIVE, SKETCH-AC and STOCHASTIC are between 11.9 to 17.1. This comparison is not particularly fair, since both CVC4-1.5.1 and EUSOLVER solved more benchmarks in general, so that might be the reason. For this reason we give the exact size expression per benchmark in the detailed evaluation figures (Figs. 6 to 9). Looking at these figures we can see that in many instances where the benchmark was solved by both CVC4-1.5.1 and EUSOLVER, the size of the expression generated by EUSOLVER was in a smaller bucket according to our pseudo-logarithmic scale, see for instance the `array_search*` benchmarks and the `fg_max*` benchmarks.

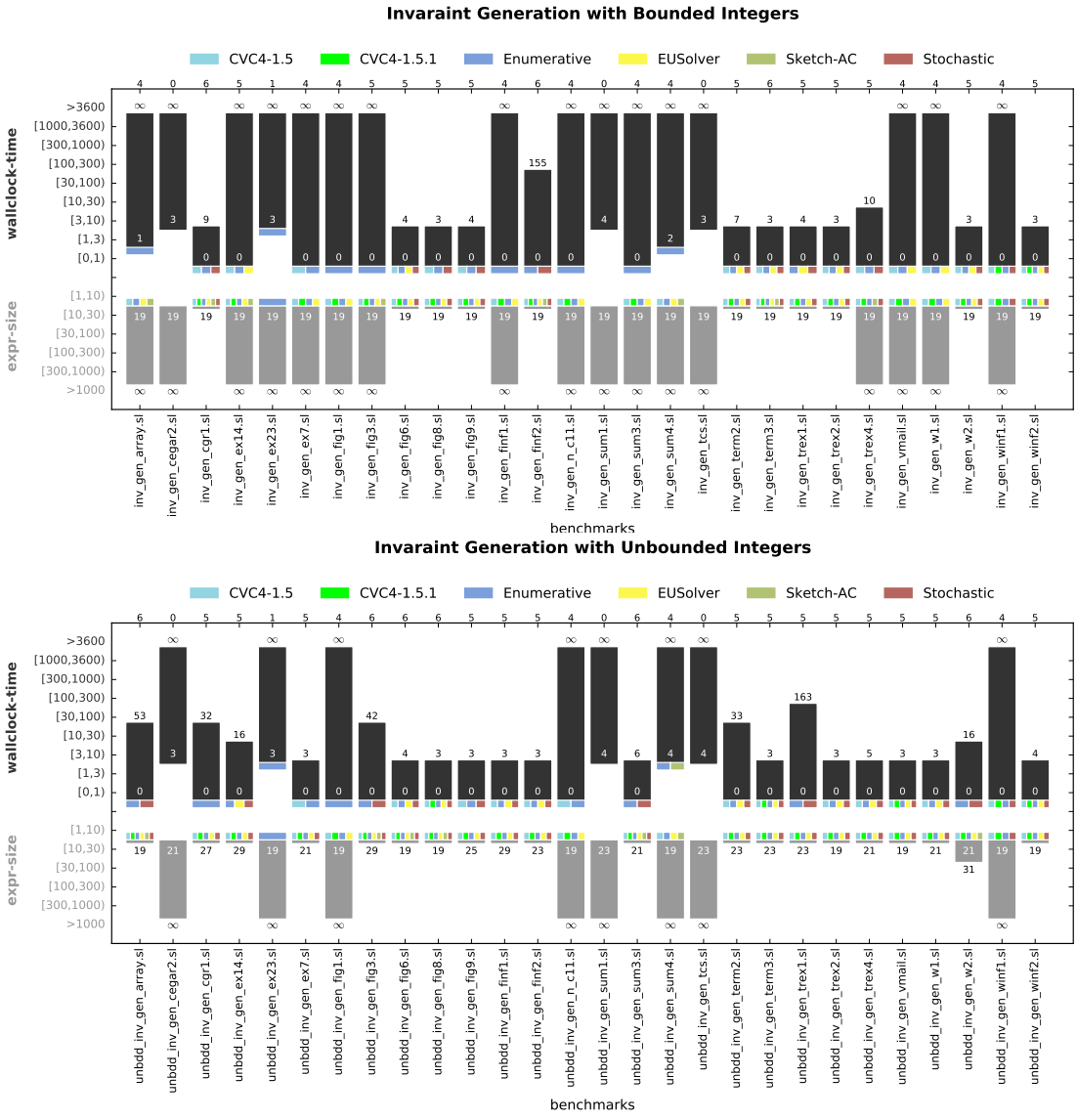

Figure 7: Evaluation of Invariant benchmarks of the general track.

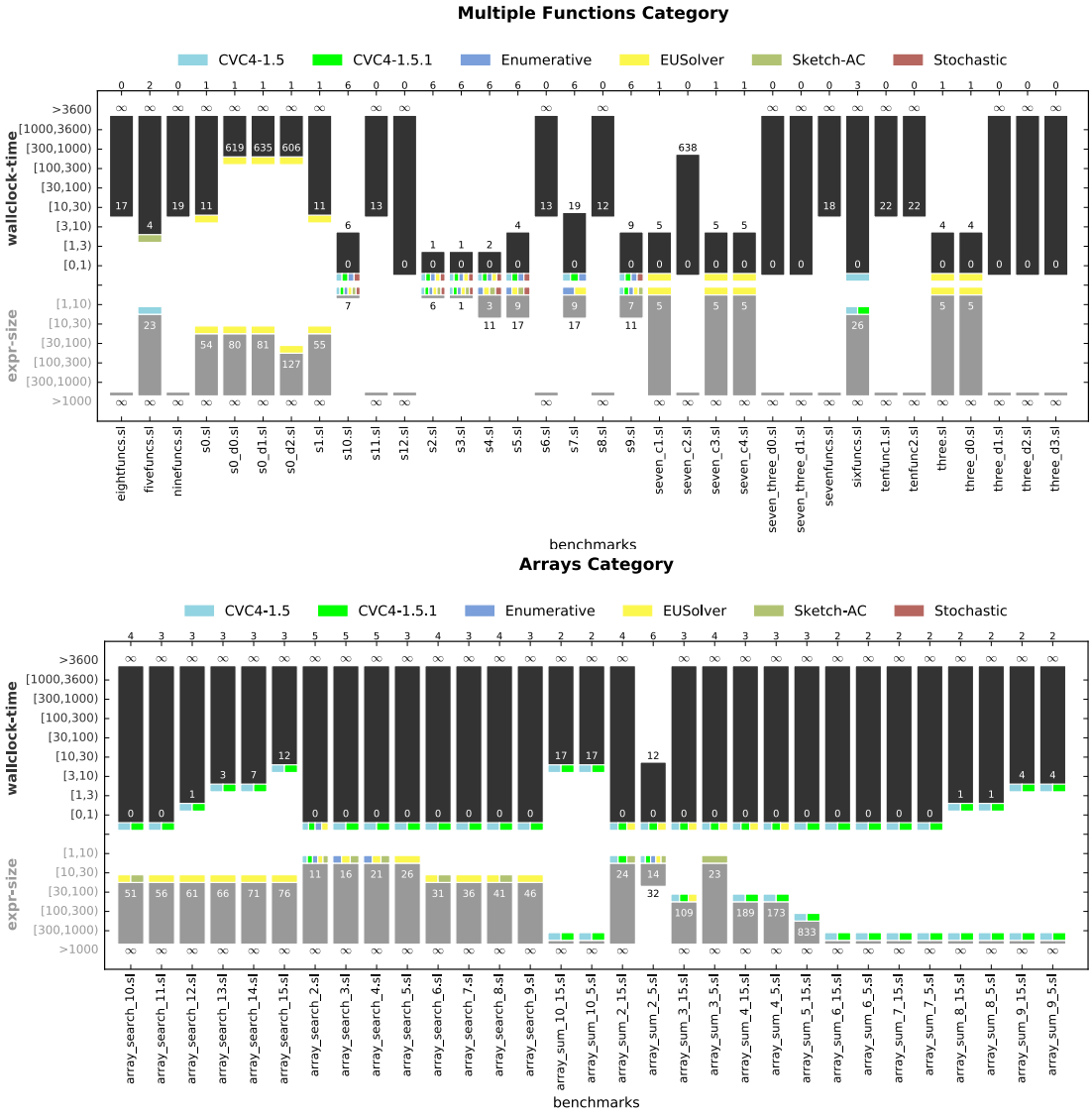

Figure 8: Evaluation of Multiple Functions and Arrays benchmarks.

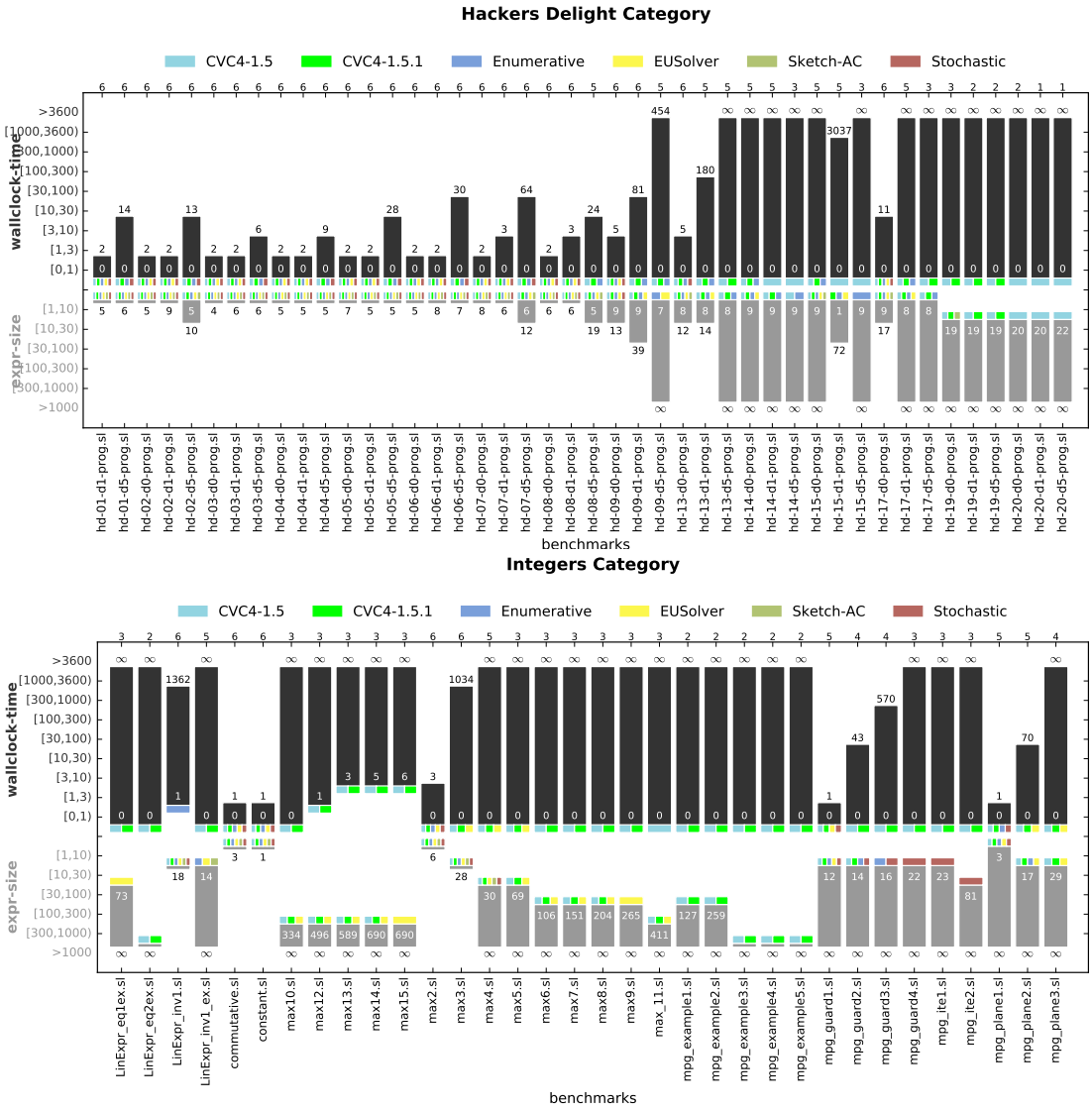

Figure 9: Evaluation of Hackers Delight and Integer benchmarks.

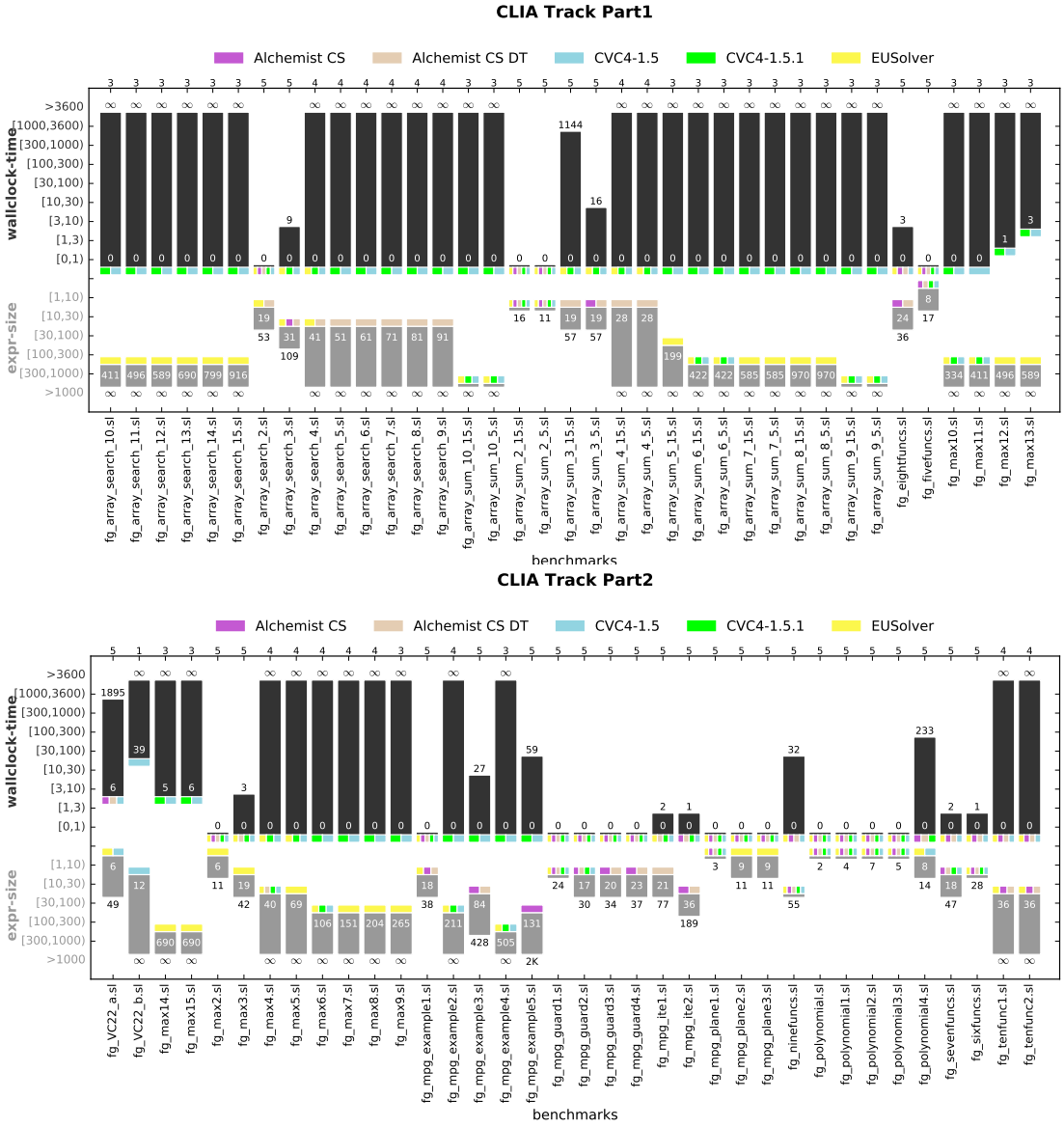

Figure 10: Evaluation of CLIA track benchmarks.

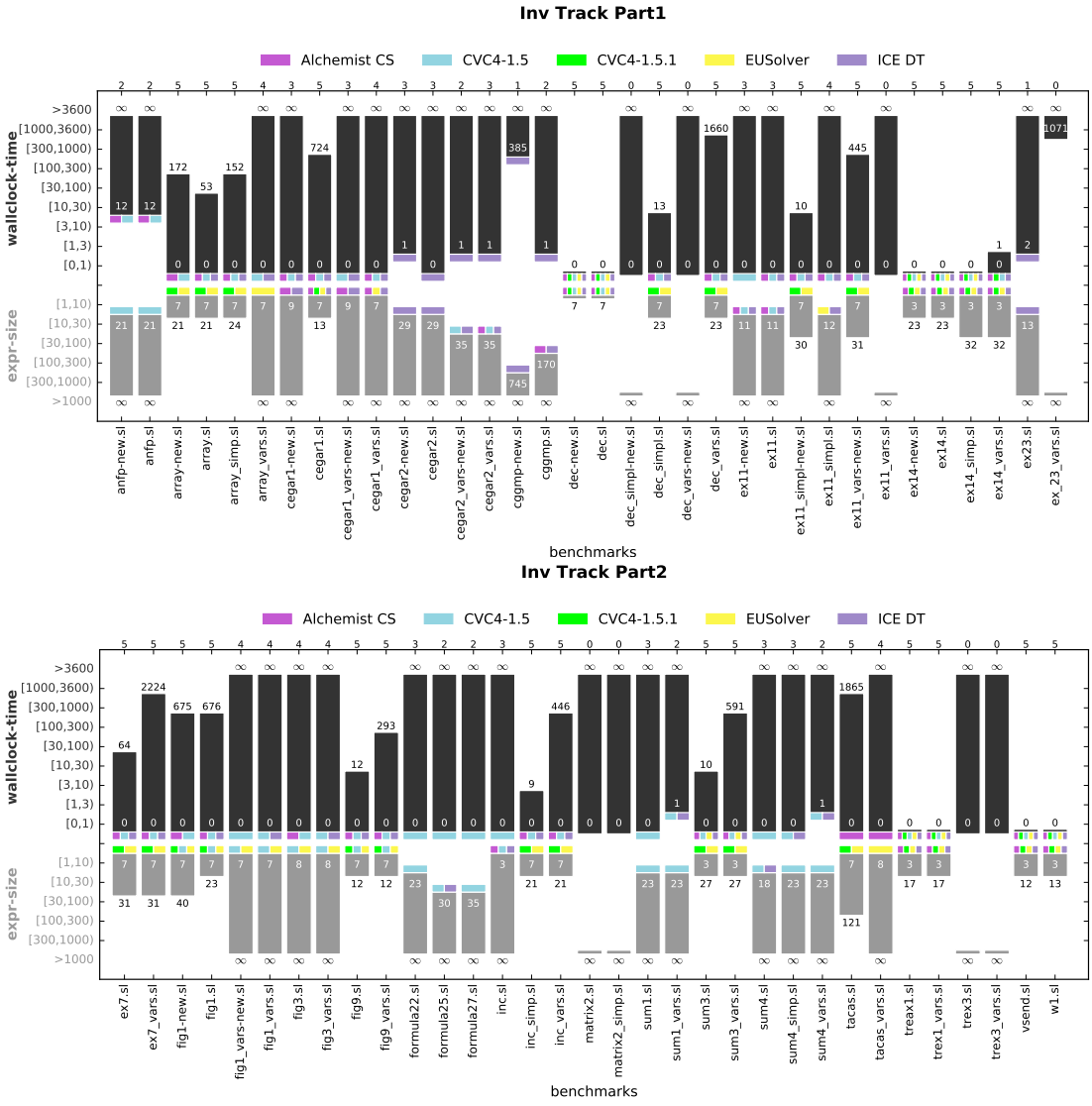

Figure 11: Evaluation of Invariant track benchmarks.

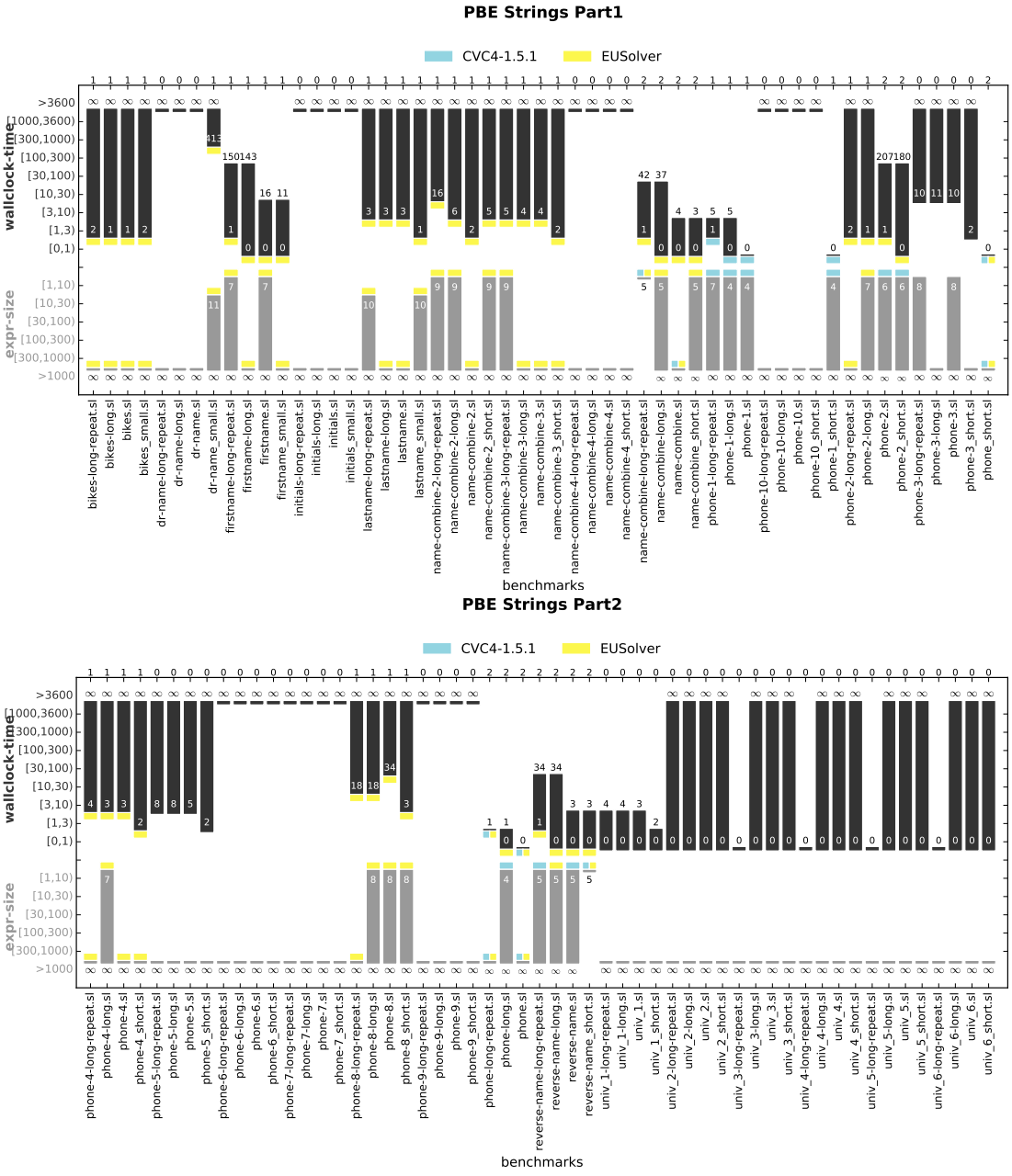

Figure 12: Evaluation of PBE-Strings benchmarks.

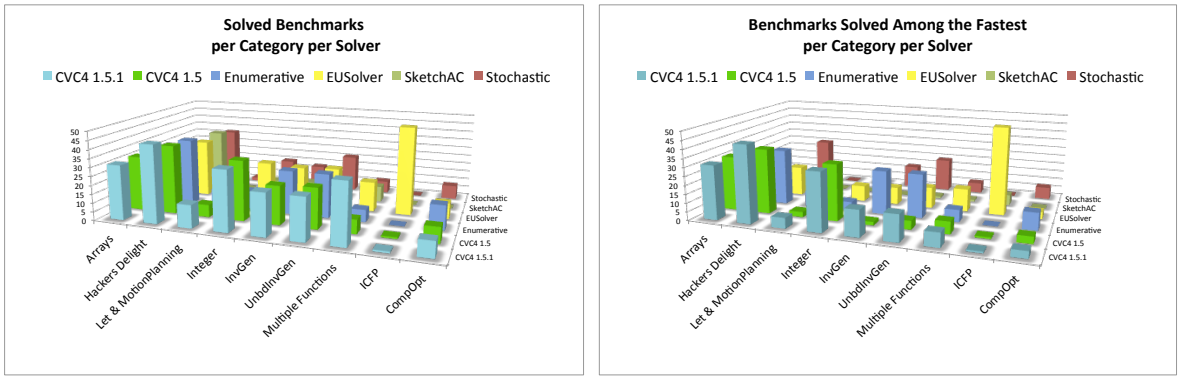

Figure 13: Results of General Tracks per Solver per Category.

4 Discussion

We present a few interesting dimensions in which the SyGuS competition has evolved over the past 3 years. The timeline for the tracks and the solvers submitted for each competition is shown in Figure 14. The first competition in 2014 had a single General track, and 5 solvers competed in the competition that included enumerative, stochastic, symbolic, and machine learning-based synthesis algorithms. The second competition introduced two new tracks: conditional linear integer arithmetic track and the invariant synthesis track. There were 7 new solver submissions that implemented SMT-based quantifier instantiation, adaptive concretization of unknowns, BDD-based symbolic algorithms, and geometric optimization based synthesis algorithms. In the 2016 competition, we introduced another new track, the PBE track, and two new solvers EUSOLVER and CVC4-1.5.1 participated in the competition.

The percentage of benchmarks in the General track solved by the solvers participating in the first competition as compared to the solvers in the third competition is shown in Figure 15. As we can observe, a much higher fraction of benchmarks are solved by solvers in the third competition as compared to the solvers from the first competition. The successful and challenging classes of benchmarks from each competition is shown in Figure 16. We can observe that many of the challenging benchmarks from the previous competition are tackled by the solvers in the newer competition.

Acknowledgments

We would like to thank the following people for various interesting discussions related to the competition, its tracks, the SyGuS format and various other topics related to syntax-guided synthesis: Viktor Kuncak, Arjun Radhakrishna, and Andrew Reynolds.

We would like to thanks the StarExec [34] team, and especially Aaron Stump, for allowing us to use their platform and for their remarkable support for SyGuS-Comp's special needs.

This research was supported by US NSF grant CCF-1138996 (ExCAPE).

References

[1] Takuya Akiba, Kentaro Imajo, Hiroaki Iwami, Yoichi Iwata, Toshiki Kataoka, Naohiro Takahashi, Micha Moskal & Nikhil Swamy (2014): *Calibrating Research in Program Synthesis Using 72,000 Hours of Programmer Time*.

1

	1st Comp.	2nd Comp.	3rd Comp.
Tracks:	**General**	General **CLIA** **Inv**	General CLIA Inv **PBE**
Solvers:	Enumerative Symbolic **Sketch** Stochastic **Alchemist**	Enumerative **SosyToast** **Sketch$_{AC}$** Stochastic **Alchemist$_{CS}$** **Alchemist$_{CSDT}$** **ICE** **CVC4 1.5**	Enumerative Sketch$_{AC}$ Stochastic Alchemist$_{CS}$ Alchemist$_{CSDT}$ ICE **CVC4 1.5.1** **EUSolver**

Figure 14: The timeline for tracks and solvers for each competition.

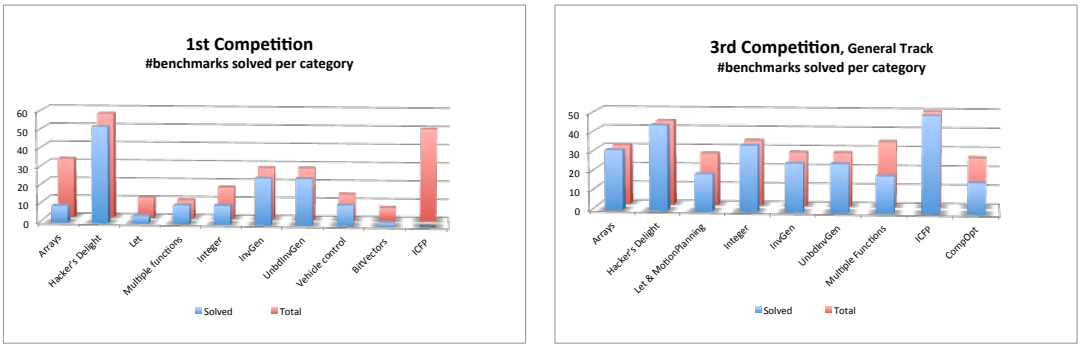

Figure 15: The comparison between the number of benchmarks solved out of the total in the 1st competition and the general track of the 3rd competition.

1

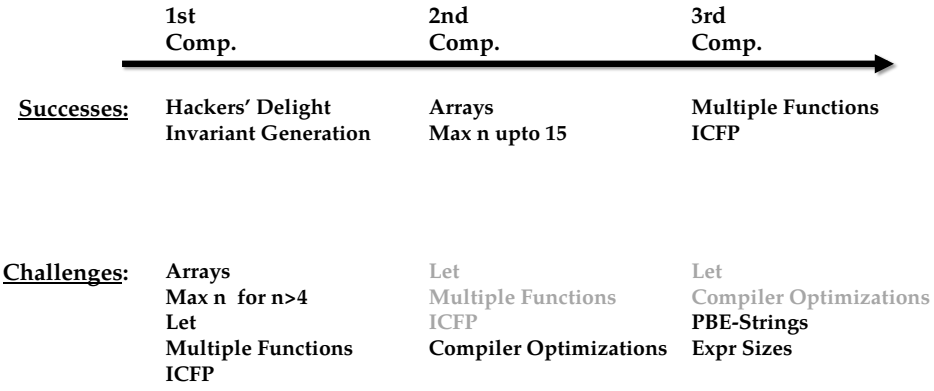

Figure 16: The timeline for successful and challenging classes of benchmarks.

[2] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak & Abhishek Udupa (2015): *Syntax-Guided Synthesis*. In: *Dependable Software Systems Engineering*, IOS Press, pp. 1–25, doi:10.3233/978-1-61499-495-4-1.

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

[4] Rajeev Alur, Pavol Cerný & Arjun Radhakrishna (2015): *Synthesis Through Unification*. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pp. 163–179, doi:10.1007/978-3-319-21668-3_10.

[5] Rajeev Alur, Dana Fisman, P. Madhusudan, Rishabh Singh & Armando Solar-Lezama: *SyGuS Syntax for SyGuS-COMP15*.

[6] Rajeev Alur, Dana Fisman, Rishabh Singh & Armando Solar-Lezama: *SyGuS Syntax for SyGuS-COMP16*.

[7] Rajeev Alur, Dana Fisman, Rishabh Singh & Armando Solar-Lezama (2015): *Results and Analysis of SyGuS-Comp'15*. In: *SYNT*, EPTCS, pp. 3–26, doi:10.4204/EPTCS.202.3.

[8] Sarah Chasins & Julie Newcomb (2016): *Using SyGuS to Synthesize Reactive Motion Plans*. In: *5th Workshop on Synthesis, SYNT 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*.

[9] Hassan Eldib, Meng Wu & Chao Wang (2016): *Synthesis of Fault-Attack Countermeasures for Cryptographic Circuits*. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pp. 343–363, doi:10.1007/978-3-319-41540-6_19.

[10] John K. Feser, Swarat Chaudhuri & Isil Dillig (2015): *Synthesizing data structure transformations from input-output examples*. In: *PLDI*, pp. 229–239, doi:10.1145/2737924.2737977.

[11] Pranav Garg, Christof Löding, P. Madhusudan & Daniel Neider (2014): *ICE: A Robust Framework for Learning Invariants*. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as*

Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, pp. 69–87, doi:10.1007/978-3-319-08867-9_5.

[12] Pranav Garg, Daniel Neider, P. Madhusudan & 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 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 499–512, doi:10.1145/2837614.2837664.

[13] Sumit Gulwani (2011): *Automating string processing in spreadsheets using input-output examples*. In: *POPL*, pp. 317–330.

[14] Sumit Gulwani, William R. Harris & Rishabh Singh (2012): *Spreadsheet data manipulation using examples*. *Commun. ACM* 55(8), pp. 97–105, doi:10.1145/2240236.2240260.

[15] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama & Jeffrey S. Foster (2015): *Adaptive Concretization for Parallel Program Synthesis*. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pp. 377–394, doi:10.1007/978-3-319-21668-3_22.

[16] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia & 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, pp. 215–224, doi:10.1145/1806799.1806833.

[17] Alan Leung, John Sarracino & Sorin Lerner (2015): *Interactive parser synthesis by example*. In: *PLDI*, pp. 565–574, doi:10.1145/2737924.2738002.

[18] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang & Daniel Jackson (2015): *Alloy*: A General-Purpose Higher-Order Relational Constraint Solver*. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pp. 609–619, doi:10.1109/ICSE.2015.77.

[19] Daniel Neider, P. Madhusudan & Pranav Garg (2015): *ICE DT: Learning Invariants using Decision Trees and Implication Counterexamples*. Private Communication.

[20] Daniel Neider, Shambwaditya Saha & P. Madhusudan (2015): *Alchemist CS: An SMT-based synthesizer for Functions in Linear Integer Arithmetic*. Private Communication.

[21] Peter-Michael Osera & Steve Zdancewic (2015): *Type-and-example-directed program synthesis*. In: *PLDI*, pp. 619–630, doi:10.1145/2737924.2738007.

[22] Mukund Raghothaman & Abhishek Udupa (2014): *Language to Specify Syntax-Guided Synthesis Problems*. *CoRR* abs/1405.5590.

[23] Veselin Raychev, Max Schäfer, Manu Sridharan & Martin T. Vechev (2013): *Refactoring with synthesis*. In: *OOPSLA*, pp. 339–354, doi:10.1145/2509136.2509544.

[24] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli & 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*, pp. 198–216, doi:10.1007/978-3-319-21668-3_12.

[25] Heinz Riener & Rudiger Ehlers (2015): *absTract sOlution Analyzing Synthesis Tool (System Description)*. Private Communication.

[26] Shambwaditya Saha, Pranav Garg & P. Madhusudan (2015): *Alchemist: Learning Guarded Affine Functions*. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pp. 440–446, doi:10.1007/978-3-319-21690-4_26.

[27] Shambwaditya Saha, Daniel Neider & P. Madhusudan (2015): *Alchemist CS DT: Synthesizing Guarded Affine Functions using Constraint Solving and Decision-tree Learning*. Private Communication.

[28] Rishabh Singh (2016): *BlinkFill: Semi-supervised Programming By Example for Syntactic String Transformations*. *PVLDB* 9(10), pp. 816–827.

[29] Rishabh Singh & Armando Solar-Lezama (2011): *Synthesizing data structure manipulations from storyboards*. In: *FSE*, pp. 289–299, doi:10.1145/2025113.2025153.

[30] Rishabh Singh & Armando Solar-Lezama (2012): *SPT: Storyboard Programming Tool*. In: *CAV*, pp. 738–743, doi:10.1007/978-3-642-31424-7_58.

[31] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík & Kemal Ebcioglu (2005): *Programming by sketching for bit-streaming programs*. In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pp. 281–294, doi:10.1145/1065010.1065045.

[32] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia & Vijay A. Saraswat (2006): *Combinatorial sketching for finite programs*. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pp. 404–415, doi:10.1145/1168857.1168907.

[33] Saurabh Srivastava, Sumit Gulwani & Jeffrey S. Foster (2010): *From program verification to program synthesis*. In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pp. 313–326, doi:10.1145/1706299.1706337.

[34] Aaron Stump, Geoff Sutcliffe & Cesare Tinelli (2014): *StarExec: A Cross-Community Infrastructure for Logic Solving*. In: *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, pp. 367–373, doi:10.1007/978-3-319-08587-6_28.

[35] Emina Torlak & Rastislav Bodík (2014): *A lightweight symbolic virtual machine for solver-aided host languages*. In: *PLDI*, p. 54, doi:10.1145/2594291.2594340.