# Optimizing SMT solving strategies by learning with an evolutionary process

Nicolás Gálvez Ramírez
Université d'Angers and Universidad Técnica Federico Santa María
Angers, France
Valparaíso, Chile
ngalvez@inf.utfsm.cl

Eric Monfroy
LS2N, UMR CNRS 6004
Université de Nantes
Nantes, France
eric.monfroy@univ-nantes.fr

Frédéric Saubion
LERIA
Université d'Angers
Angers, France
frederic.saubion@univ-angers.fr

Carlos Castro
Universidad Técnica Federico Santa María
Valparaíso, Chile
carlos.castro@inf.utfsm.cl

***Abstract*—This paper deals with program optimization, i.e., learning of more efficient programs. The programs we want to improve are Z3 solving strategies. Z3 is a SMT (SAT Modulo Theory) solver which is currently developed by Microsoft Research. We define strategy generators based on evolutionary processes. SMT solving strategies include various aspects that can affect the performance of a SMT solver dramatically.Each of these elements includes a huge amount of options which cannot be exploited without expert knowledge. We define a generic evolutionary algorithm based on genetic programming concepts. This strategy generation process aims at learning better strategies by successive improvements, using rules that can be combined in order to handle both structures and parameters of the strategies. We experiment 7 different strategies generators on 2 SMT logics (QF_LRA,QF_LIA). The results show that the learned strategies improve default strategies available in the solver.**

***Keywords : Program optimization; Genetic Programming; SAT modulo theory; Strategy Generation; Learning***

## I. Introduction

Program optimization is the process of modifying a program to make some aspects of it work more efficiently or use fewer resources (for example, a program may be optimized so that it executes more rapidly, or with less memory).

SAT Modulo Theories (SMT) is a generalization of the famous satisfaction problem (SAT) for logical formulas over one or more theories: Boolean variables may be replaced by formulas expressed over different theories (e.g., real arithmetic, arrays, ...) in order to validate a logical formula. For instance, $\forall x \exists y \ ((x > 1) \vee (y < 0)) \wedge (x + y = 0)$ is a formula over linear integer arithmetic with quantified variables (see [8] for a recent survey). Hence, an SMT solver usually needs to combine several algorithms and components in order to improve proof efficiency: a solving strategy (i.e., a program) defines how to select and use these solving components. The *SMT-LIB standard* introduces the concepts of Theories and Logics in order to classify problems: a problem belongs to a logic; a logic refers to some theories. Z3 [2] from Microsoft Research is one of the most famous SMT system.

In this paper, we want to optimize Z3 solving strategies in order to solve more problems, and if possible, more quickly. In some previous work [9], we used techniques issued from automated parameter tuning [4], [6] to optimize some given strategies. We obtained good results when restricting the scope of strategy transformation to mainly changing numerical parameters and ordering of solvers in a sequence). The strategy space is considerably huge with regards to classic parameter tuning tasks: a program must be learned/modified/optimized, some components must be selected, and some value parameters must be tuned. These tools allow users to exert strategic control over core heuristic aspects of high-performance SMT solvers, a challenge that has been pointed out in [3].

We thus propose a strategy generation process based on evolutionary computation techniques that aim at learning better strategies by successive improvements. This automated strategy generation process has to take into account the following components:

1) an evolutionary algorithmic engine that generates and evaluates strategies in order to converge to a strategy that improves solver performances on a given set of SMT instances (mainly in terms of number of solved instances in a given time, i.e., in the conditions of the SMT Comp [1]) ;
2) a set of rules that are used to build and/or modify strategies within the previous algorithmic engine; some of these rules modify the structure of the strategies (i.e., change the program) whereas other rules focus on parameter values management;
3) an initial strategy since the search space is huge and starting from scratch may often be intractable; here, we start from Z3 default strategies;

In this paper, we propose a generic evolutionary algorithm. Its components (and the rules it can manage) are then instantiated in order to obtain several strategy gener-

978-1-5386-7879-4/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCS.2018.00132

ators/optimizers. We experimented 7 generators on 2 SMT logics (QF_LRA,QF_LIA). The results show that the learned strategies improve initial strategies.

## II. Z3 Solving Strategy Language

The Z3 solving strategy language is defined by a grammar whose terminals may be: solvers to check the satisfiability of a formulas over some theories, probes to check properties of formulas, heuristics to split a problem into a sequence of sub-problems, and (numeric or symbolic value) parameters to control the behaviour of strategies. These components are linked by operators such as `try-for` or `using-params` to assign parameters to sub-strategies (for instance a running time or a random seed), and combinators such as `and-then` (a kind of conjunction of the results of sub-strategies) and `or-else` (a kind of disjunction of the result of sub-strategies) for combining sub-strategies. Fig. 1 shows a strategy with solvers (e.g., `sat` or `smt`), a probe (`fail-if(not(is-ilp))`), a heuristic (`simplify`), and combinors (`or-else` and `and-then`).

```
(and-then
        (fail-if (not(is-ilp)))
        simplify
        split-clause
        (or-else
                (try-for sat 100)
                (using-params
                        smt
                :random-seed 100)))
```

Fig. 1. Example: a Z3 solving strategy.

The application of a strategy to an instance returns: *no* when the strategy proves that there is no solution, a solution when the strategy finds one, and *unknown* otherwise. These tools allow users to exert strategic control over core heuristic aspects of high-performance SMT solvers, a challenge that has been pointed out in [3]. Therefore, due to the complexity of this strategy language, there is a clear need of automated processes to help the users to design efficient strategies for given SMT problems.

## III. Strategy Generation Algorithm

We propose a classic evolutionary process [5] described by Algorithm 1. The evolution loop is applied on individuals of the population, i.e., on Z3 solving strategies. This is a steady-state evolutionary algorithm: at each iteration, an evolution rule is chosen. The individuals (as many as required by the rule) are selected and processed by the chosen rule. The resulting individual(s) are then classically inserted in the population.

$Ind^n$ is a vector of $n$ individuals (remind that individuals are strategies). At each iteration, the rule selection function $select_R$ selects a rule from possible set of variation rules (described later). Since these rule may apply on different elements of the strategy, we have to identify the arity of the rule $ar(r)$. Therefore, the individual selection function

**Algorithm 1** Evolutionary Algorithm Scheme

**Input:** a SMT-LIB logic set of instances,
an initial strategy $Is$,
a population size $N$,
a set of evolution rules $R$,
a rule selection function $select_R$,
an individual selection function $select_I$,
a fitness function *fitness*,
an ending criterion $end_C$
**Output:** Optimized strategy $st^*$

```
Initialize population using st
repeat
    r ← select_R(R)
    ar(r) = n
        Ind^n ← select_I(n, population, fitness)
        Ind^n →_r Ind'^n'
    insert( Ind'^n' ,population,fitness)
    st* ← best(population)
until end_C
return st*
```

($select_I$) and the insertion function ($insert$) have to take into account this variability. These functions may be adapted to fit a particular strategy generator and are detailed later.

We now briefly describe the main components of our algorithm

- Initialization of the population: $Is$ provides an initial strategy. The initial population is generated with $size$ individuals built from $Is$ with a randomly chosen rules from $SV$.
- Fitness function : the fitness function involves the number of solved instances and, as a second criterion, the time used for solving these instances. We define our fitness function as:

$$\begin{aligned} f &: PStrat \mapsto \mathbb{N} \times \mathbb{N} \\ f(st) &= (i(st), t(st)) \end{aligned} \tag{1}$$

  where:
  1) $i(st)$: number of instances solved using the strategy $st$.
  2) $t(st)$: elapsed time for solving these instances.

  Since this fitness function is defined on $\mathbb{N}\times\mathbb{N}$, we use the lexicographic ordering $\succ \equiv (>,>)$ in order to compare fitness values, i.e., given $st, st' \in PStrat, f(st) = (i,t), f(st') = (i',t')$, we have $f(st) \succ f(st')$ if and only if $i > i'$, or $i = i'$ and $t < t'$.
- Individuals selection: the choice of the individual is performed by a classic tournament selection that consists in selecting randomly $k$ individuals of the population and then, selecting the best $n$ individuals according to the fitness function (see [4] for more details on evolutionary algorithms). As already mentioned, $n$ is adjusted according to the arity of the rule to be applied.
- Individual insertion: insertion consists in replacing the current worst $n'$ individuals of the population by the

generated individuals. Note that again $n'$ depends on the selected rule.

- Ending criterion: it corresponds either to a time limit or to the fact that the population has converged, i.e., 80% of the population has the same fitness.

The initial population is generated with $size$ individuals built from the $Is$ initial strategy with randomly chosen variation rules - to get some diversity. The fitness function involves the number of solved instances and, as a second criterion, the time used for solving these instances. Ending condition corresponds to a limited time or the observed convergence of the population.

The rules constitute the key feature of our strategy generation process. A rule is applied on a strategy in order to modify either its structure or its behavior. According to the Z3 strategy grammar, we classify these possible modifications according to the elements that they affect:

1) structural components: the tactics (i.e., solvers, probes, and heuristics) that can be used in strategies and the combinators that define how tactics are combined and applied;
2) behavioral parameters: the parameters values (either numeric or symbolic) that help to define the specific behavior of the tactics (execution time, seeds, etc.).

Intuitively, structural rules modify the structure of a strategy (e.g., by changing a combinator, by changing the arity of a combinator, by adding or removing a combinator, etc.). They can also change the tactics, e.g., replacing a heuristic by another one. Behavioral rules change parameters values, e.g., to give more time to a given solver, to change a random seed, or to change a solver parameter.

According to previous remarks, we can define more precisely 4 sets of rules.

Structural variation rules ($\mathbf{SV}$): these rules modify the structure of the strategy by introducing or modifying combinators, solvers or heuristics. For instance, a strategy `and-then(simplify,try-for sat 100)` can be transformed into `and-then(simplify, try-for smt 100)` by changing the solver or into `and-then(simplify,split-clause,try-for sat 100)` by introducing a new heuristics. Note that some constraints have to be checked to insure that the strategy is correct according to Z3 requirements.

Structural recombination rules ($\mathbf{SR}$): these rules are used to exchange subparts of strategies. Taking two individuals as input, this is a classic mechanism in genetic programming [7] that allows the algorithm to produce two new individuals. Again, the coherence of this swap has to be insured by selecting compatible sub-strategies.

Behavioral variation rules $\mathbf{BV}$ : these rules focus on the parameters of the strategy. In Z3, numerous parameters can be used for the different components (some solvers have more than 50 parameters). When these parameters are not explicitly mentioned in a strategy, default values are used. In order to limit the size of our search space, we do not consider introduction of new parameters and consider only parameters from initial strategies. Moreover, time parameters are introduced to manage the time budget allocated to a given strategy according to user's needs using the `try-for` operator. These time values are carefully handled in our process to finely tune them.

Behavioral recombination rules $\mathbf{BR}$: we introduce a specific rule to better diversify parameters values. This rule is applied on the whole population and uniformly select parameters values to get a new strategy.

These different sets of rules we now design different algorithms by considering either behavioral rules or structural rules or even by combining them in hybrid strategy generators. We now present some possible combinations.

## IV. Selected Strategy Generators

We consider 7 strategy generators: 2 simple generators, and 5 hybrid generators. Back to Algorithm 1, the different generators involve different sets of rules as well as different rule selection functions $select_R$. A concise view of the respective designs of these generators is proposed in Table I, which shows the set of rules used in the generators together with their proportion of use. Note that $+$ indicates that rules are used in sequence, $^*$ stands for a loop on a sequence, $|$ indicates a possible choice between different rule sets, and ; the sequential composition of two sub-processes. There are 2 vectors of proportions when the hybrid generator is a sequence of 2 generators.

TABLE I
Rules and their use proportion inside the generator

| Gen. | Structural | | Behavioural | | Use and Proportion |
|---|---|---|---|---|---|
| | $SV$ | $SR$ | $BV$ | $BR$ | |
| StrucG (structure) | ✓ | ✓ | ✗ | ✗ | $(SR+SV)^*$ 2:1:0:0 |
| ParG (parameters) | ✗ | ✗ | ✓ | ✓ | $(BR+BV)^*$ 0:0:1:1 |
| LocG (local search) | ✗ | ✗ | ✓ | ✗ | $BV^*$ 0:0:1:0 |
| Struc+ParG | ✓ | ✓ | ✓ | ✓ | StrucG; ParG (2:1:1:1; 18:10:3:1) |
| Struc+LocG | ✓ | ✓ | ✓ | ✓ | StrucG; LocG (2:1:1:0; 18:10:3:0) |
| H1 | ✓ | ✓ | ✓ | ✓ | $(SR\|SV)^*;(BR\|BV)^*$ (2:1);(1:1) |
| H2 | ✓ | ✓ | ✓ | ✓ | $(SR\|SV)^*;(BR\|BV)^*$ reactive/adaptive |

**StrucG** is a classic genetic programming procedure which starts from a given initial strategy to learn the best possible strategy by applying structural rules ($S$). The sequence of rule application is the following: a $SR$ rule is selected and applied on 2 individuals (selected using a classic tournament selection that randomly picks $n$ individuals of the population and then choose the best one w.r.t. the fitness function (see [4] for more details). We perform this action twice to get two individuals.) leading to two new individuals (i.e., a kind of crossover rule). The 2 next loops apply $SV$ rules over the newly generated

individuals. The insertion function replaces the current worst individual of the population by the best generated individual using the $SR$ rule, and it replaces the second worst individual of the population if the best generated individual using $SV$ outperforms it. The ending criterion is either a time limit (i.e., the $Ltb$ parameter) or a convergence property (i.e., 80% of the population has the same fitness).

Given a strategy, **ParG** searches for the best possible parameter configuration using Behavioral rules (set $B$). The given initial strategy is analyzed in order to create an initial population which size is given by the largest number of value a parameter can get. The rule selection function selects a rule from $BR$ which is applied on some randomly selected individuals. Then, (at the next loop) a rule from $BV$ is selected and applied on the result of the previous rule application. The insertion function works as in StrucG, but with regards to $B$R and $BV$ sets of rules. The ending criterion is the same as in StrucG.

**LocG** can be seen as an evolutionary algorithm with one individual. Thus, only variation rules are used. LocG procedure starts from a given initial strategy and applies only rules from $SV$ in order to explore its neighborhood. In our case, the neighborhood of a strategy is thus fully defined by all possible values changes of a single parameter. We refer the reader to [6] for more details on local search.

The two first hybrid generators correspond to off-line integration: two algorithms are executed sequentially. First, a structural rule based algorithm is executed in order to obtain an optimized strategy with regards to structure modification. Then, its result is the initial strategy of a "parameter tuning algorithm", i.e., a behavioral rule based algorithm. The two hybrids are: 1) **Struc+ParG** which is a genetic programing structural algorithm (StrucG) calibrated with an evolutionary programming algorithm (ParG), and 2) **Struc+LocG** which is a genetic programing structural algorithm (StrucG) calibrated with a local search algorithm (LocG).

**H1** is an on-line cooperation, based on the engine of StrucG but using all the rules ($SV \cup SR \cup BV \cup BR$). The rule selection repetitively performs the following sequence: it first selects a rule from $SR$ (to be applied on 2 selected members of the population) that generates 2 individuals on which 2 structural variation rules ($SV$) are applied (1 rule on 1 individual); a behavioural recombination ($BR$) rule then tries to improve the best result obtained with the previous structural variation rules; finally, a behavioural variation ($BV$) rule tries to improve the best result. The insertion function is the same as in StrucG.

**H2** is based on H1. The main difference is the rule selection process. Two parameters, $\phi$ and $\rho$ (with $\phi < \rho$), are required: $\phi$ indicates how many rules from $S = SV \cup SR$ are used sequentially, and $\rho - \phi$ indicates the number of rules from $B = BV \cup BR$ that are then applied. For rules of $S$, the following sequence is respected: one rule from $SR$ followed by 2 rules of $SV$ applied on the 2 resulting individuals. For behavioral rules, the sequence consists first of a behavioral recombination rule ($BR$) to try to improve the best individual of the population, and then a behavioral variation rule ($BV$) to try to improve the best population member. When the $\rho$ limit is reached, the counter is reset, and thus, structural rules are applied again. If a rule from $S$ has been picked, the insertion is the one from StrucG; otherwise, the best individual obtained with the selected rules from $BR$ and $BV$ replaces the worst individual of the population.

## V. Experimental Results And Conclusions

Our experiments involve two phases. The learning phase uses a subset of the problems to be solved to learn better strategies by means of generators. Once an optimized strategy has been built, its efficiency is evaluated, as usual for cross validation, on the whole set of problem instances.

We focus here only on two logics from Quantifier Free Linear Arithmetic logic family, which have many important SMT applications. Instances that have been solved in SMT-COMP are called *known* (either instances proved to be false by the strategy or for which a solution has been computed), and the others *unknown* . (`QF_LRA`) instances corresponds to closed linear formulas in linear real arithmetic. QF means that we conider unquantified linear real arithmetic. These instances are Boolean combinations of inequations between linear polynomials over real variables. (`QF_LIA`) instances are thus Boolean combinations of inequations between linear polynomials over integer variables.

(`QF_LIA`) has 5893 known instances and 302 unknown while (`QF_LRA`) has 1626 known instances and 56 unknown. We limit the learning time for the generators to 2 days: this is rather short with respect to the total time for trying to solve all the instances.

In Table II, the different strategy generators are compared with regards to the number of instances they succeed to solve and their solving times. The Z3 default line corresponds to the performance of the Z3 default strategy that is used as input for the generators. The two columns % solv and % time correspond to the improvements with regards to the default strategy. Of course, solving time must be used as a second criterion (since it is related to the number of solved instances). The strategies are evaluated with two different time budgets for each instance : 10 seconds and 2400 seconds (as in SMT competitions).

A first important point is that we succeed in improving the default Z3 strategy for QF_LIA known: indeed, this logic corresponds to the set of instances for which the default strategy has been finely designed by experts, using an empirical and costly process. Moreover, we also see that our generators allow us to improve strategies for the other cases. The generated strategies are able to solve new instances (in particular in unknown sets). Comparing the different generators, we may observe that the hybrid approaches seem to constitute a rather good compromise by mixing structural improvements of the strategy with finer tuning of the parameters. Note that we get similar results (not reported here) for others logics families (LRA and LIA).

TABLE II
EXPERIMENTAL RESULTS ON SELECTED BENCHMARKS

| Time | 10 | | | | 2400 (smt-comp) | | | |
|---|---|---|---|---|---|---|---|---|
| QF_LIA known | solved | time | % solv | % time | solved | time | % solv | % time |
| Z3 default | 4102 | 22689,20 | - | - | 5617 | 659286,39 | - | - |
| ParG | 4104 | 23194,00 | 0,05 | 2,22 | 5508 | 928736,69 | -1,94 | 40,87 |
| LocG | 4090 | 22992,18 | -0,29 | 1,34 | 5522 | 894412,13 | -1,69 | 35,66 |
| StrucG | 4220 | 22957,21 | 2,88 | 1,18 | 5536 | 831662,86 | -1,44 | 26,15 |
| Struc+ParG | 4097 | 24368,00 | -0,12 | 7,40 | 5598 | 693616,42 | -0,34 | 5,21 |
| Struc+LocG | 3977 | 24363,48 | -3,05 | 7,38 | 5489 | 942000,73 | -2,28 | 42,88 |
| H1 | 4209 | 20956,49 | 2,61 | -7,64 | 5630 | 633432,62 | 0,23 | -3,92 |
| H2 | 4102 | 22684,11 | 0,00 | -0,02 | 5605 | 710597,18 | -0,21 | 7,78 |
| QF_LIA unknown | solved | time | % solv | %time | solved | time | % solv | % time |
| Z3 default | 110 | 2120,39 | - | - | 130 | -298769,18 | - | - |
| ParG | 197 | 1185,68 | 79,09 | -44,08 | 208 | -480806,91 | 60,00 | 60,93 |
| LocG | 188 | 1236,72 | 70,91 | -41,67 | 197 | -458378,72 | 51,54 | 53,42 |
| StrucG | 198 | 1081,21 | 80,00 | -49,01 | 210 | -490382,22 | 61,54 | 64,13 |
| Struc+ParG | 203 | 1031,21 | 84,55 | -51,37 | 210 | -488629,23 | 61,54 | 63,55 |
| Struc+LocG | 199 | 1049,46 | 80,91 | -50,51 | 211 | -487799,89 | 62,31 | 63,27 |
| H1 | 199 | 1187,69 | 80,91 | -43,99 | 209 | -486727,99 | 60,77 | 62,91 |
| H2 | 199 | 1372,72 | 80,91 | -35,26 | 211 | -489968,73 | 62,31 | 64,00 |
| QF_LRA known | solved | time | % solv | %time | solved | time | % solv | % time |
| Z3 default | 1173 | 5024,45 | - | - | 1530 | 346598,57 | - | - |
| ParG | 1160 | 5248,78 | -1,11 | 4,46 | 1505 | 399381,88 | -1,63 | 15,23 |
| LocG | 1160 | 5248,78 | -1,11 | 4,46 | 1505 | 399381,88 | -1,63 | 15,23 |
| StrucG | 1250 | 4501,67 | 6,56 | -10,40 | 1571 | 184432,72 | 2,68 | -46,79 |
| Struc+ParG | 1227 | 4744,60 | 4,60 | -5,57 | 1568 | 199240,73 | 2,48 | -42,52 |
| Struc+LocG | 1227 | 4744,60 | 4,60 | -5,57 | 1568 | 199240,73 | 2,48 | -42,52 |
| H1 | 1240 | 4606,86 | 5,71 | -8,31 | 1577 | 175574,16 | 3,07 | -49,34 |
| H2 | 1259 | 4438,26 | 7,33 | -11,67 | 1571 | 184974,65 | 2,68 | -46,63 |
| QF_LRA unknown | solved | time | % solv | %time | solved | time | % solv | % time |
| Z3 default | 0 | 560,00 | - | - | 2 | 132486,58 | - | - |
| ParG | 0 | 560,00 | - | - | 2 | 131586,23 | 0,00 | -0,68 |
| LocG | 0 | 560,00 | - | - | 2 | 131586,23 | 0,00 | -0,68 |
| StratEVO | 1 | 555,47 | 1,79 | -0,81 | 32,3 | 76758,47 | 57,68 | -42,81 |
| StrucG | 1 | 554,14 | 1,79 | -1,05 | 34 | 72965,82 | 1600,00 | -44,93 |
| Struc+ParG | 1 | 554,95 | 1,79 | -0,90 | 31 | 79865,59 | 1450,00 | -39,72 |
| Struc+LocG | 1 | 555,18 | 1,79 | -0,86 | 31 | 79865,59 | 1450,00 | -39,72 |
| H1 | 1 | 553,91 | 1,79 | -1,09 | 3 | 130285,20 | 50,00 | -1,66 |
| H2 | 1 | 557,35 | 1,79 | -0,47 | 36 | 70096,20 | 1700,00 | -47,09 |

## REFERENCES


[1] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, pages 243–277, 2013.

[2] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008.*, LNCS, pages 337–340. Springer, 2008.

[3] Leonardo de Moura and Grant Olney Passmore. The Strategy Challenge in SMT Solving. In *Automated Reasoning and Mathematics*, LNCS, pages 15–44. Springer, 2013.

[4] A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[5] Agoston E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[6] Holger H. Hoos. *Automated Algorithm Configuration and Parameter Tuning*, pages 37–71. Springer Berlin Heidelberg, 2012.

[7] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[8] David Monniaux. A survey of satisfiability modulo theory. In *Computer Algebra in Scientific Computing - 18th International Workshop, CASC 2016, Bucharest, Romania, September 19-23, 2016, Proceedings*, pages 401–425, 2016.

[9] N. Gálvez Ramírez, Y. Hamadi, E. Monfroy, and F. Saubion. Evolving smt strategies. In *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 247–254, Nov 2016.