# A Framework for Autonomous Generation of Strategies in Satisfiability Modulo Theories

Nicolas Galvez Ramirez

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

université angers

EX UMBRA IN SOLEM

# THESE DE DOCTORAT DE

L'UNIVERSITE D'ANGERS

COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
*Mathématiques et Sciences et Technologies de l'Information et de la Communication*
Spécialité: Informatique

ET

L'UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

DEPARTAMENTO DE INFORMATICA, VALPARAISO, CHILI

Par

**Nicolás Sebastián GÁLVEZ RAMÍREZ**

## A Framework for Autonomous Generation of Strategies in Satisfiability Modulo Theories

**Thèse présentée et soutenue à Valparaíso, Chili, le 19/12/2018**
**Unité de recherche: LERIA (EA 2645)**
**Thèse N° : 131230**

**Rapporteurs avant soutenance :**

Christophe RINGEISSEN, Charge de Recherche Classe Normale, INRIA Nancy, Grand Est
Broderick CRAWFORD, Professeur des universités, Pontificia Universidad Católica de Valparaíso, Chili

**Composition du Jury :**

María Cristina RIFF, Professeur des universités, Universidad Técnica Federico Santa María.
Président
Christophe RINGEISSEN, Charge de Recherche Classe Normale, INRIA Nancy, Grand Est.
Broderick CRAWFORD, Professeur des universités, Pontificia Universidad Católica de Valparaíso, Chili.
Frédéric SAUBION, Professeur des universités, Université d'Angers.
Directeur de thèse
Eric MONFROY, Professeur des universités, Université de Nantes.
Carlos CASTRO, Professeur des universités, Universidad Técnica Federico Santa María.
Co-encadrants de thèse

**Invité(s)**
Hernán Astudillo, Professeur des universités, Directeur Doctorado en Ingeniería Informática, Universidad Técnica Federico Santa María.

# Acknowledgements

Random events not only define stochastic algorithms, but real life. This unpredictable adventure is comming to an end. From the beginning in Valparaíso, trying to find what to do with my scientific career, my unforgettable two years in France, and my last 2 years in Santiago: it has been hard but good.

I want to thanks my mentor Carlos Castro for opening the doors of the academia and science in my life, and allowing me to do this long journey. To my french advisors, Fred and Eric, whose (almost parental) support was fundamental to continue this path far away from my family and friends. Also, to Eric's wife: Veronica, for helping us (me and my wife) in our journey and adaptation in France.

Thanks to Hugo Traverson, Aymeric Le Dorze and Arthur Chambon for their sincere friendship and hospitality. Thanks also to Renato Markele, my brazilian partner, who finished his doctoral studies. Special thanks for Sue, Lizzie and Christina, the small french study group, I have so many good memories with all of you. Additionally, I want to thank Fabienne and Pierre for being sensational with us. I hope to see all of you soon.

Undoubtedly, my return to Chile it would not have been the same if Cristopher, Alex, Esteban, Roberto, Nicolás, Felipe, María José, Camila and Natalia, the research team of Informática in Campus San Joaquín, would not have been there for me. You have been very important to me in the last two years. I hope this goes on.

I am thankful to for the support of my doctoral colleagues Nicolás, Sergio, Pablo, Javier, César, Mariam and Alejandro. You really helped me to make a great defense dissertation.

I am also very grateful to my parents, brothers, parents-in-laws, sister-in-law and all my family and friends for their infinite love, support and kindness.

Finally, I want to dedicate to my beloved wife, Patricia, her well-deserved place in this thesis. She has been my support, my heroine and my savior. Always bringing life and light in the shadows, in my hardest days and in my darkest ages. I love you so much, nothing of this could have been possible without you. I hope our children (who are on the way), Liliana Antonia and Alfonso Nicolás, can be as good people as you to make this world a better place. I love you and our children from infinity to beyond.

*Nicolás*

# Contents

# List of Tables

# List of Figures

# 1

# Introduction

Optimisation tools have played a fundamental role in many fields including Engineering, Science, Medicine, etc. Since its debut in Economics [1], Combinatorial Optimisation problems, and the related resolution techniques, could be seen everywhere. With respect to Informatics and Computer Science, one of the last fields to adapt its problems to this paradigm has been Software Engineering, increasing considerably its development during the last fifteen years.

The application of various optimisation techniques in order to solve specific software engineering problems and improve software performance is now a common practice. The concept of Search-Based Software Engineering (SBSE) has been introduced and led Mark Harman to define a challenge [2] entitled "*Search for strategies rather than instances*". This challenge aims at avoiding specific software engineering optimisation algorithms to solve particular instances of SBSE problems, but rather to look for more global strategies. Moreover, another related purpose is to handle efficiently new unknown problems that share properties with already identified problem classes.

As software could be understood as computation systems of instructions following a defined logic, therefore a logic can be used to characterise all the possible events of a software piece. This makes software and first-order logics related since its foundations. Therefore, automated deduction processes could be used as intermediate layer between software engineering problems and tools to solve them [3].

Satisfiability Modulo Theories (SMT) [4], a generalisation of the most famous Constraint Satisfaction problem Boolean Satisfiability Problem (SAT) [5], and its tools [6, 7, 8] have been arise as one of the most useful field to address SBSE problems, given the amount of software

related first-order logics included in their automated deduction systems. Thus, SBSE problems are one of the biggest applications in SMT as stated in [3, 4, 9].

Moreover, developers of Z3 [6], one of the most efficient SMT solvers, Leonardo de Moura and Grant Onley Passmore have defined *The Strategy Challenge in SMT* [10]: ***to build theoretical and practical tools allowing users to exert strategic control over core heuristic aspects of high-performance SMT solvers***. Where, through a defined language, end-users can control heuristics components of solver through programming, affecting dramatically the performance of a SMT solver such as Z3.

However, end-users do not have the required knowledge in order to use properly all the heuristics features in SMT solvers, therefore the absence of expert guidelines could lead to several unsuccessful attempts.

In this work we address the *Strategy Challenge in SMT* defining a framework for the generation of strategies for Z3, i.e. a practical system to automatically generate SMT strategies without the use of expert knowledge in order to improve SMT solver performance, and being able to use it in an autonomous search system for continuous improvement. Note this is very interesting for SBSE and to tackle the mentioned SBSE challenge, but also for others SMT applications including Combinatorial Optimisation problems such as Scheduling or Planning, as well as building more efficient and robust Automated Deduction tools.

## 1.1 Research Goals

The initial motivation of this research, as explained above, is the idea of founding an approach to solve different classes of search-based software engineering problems, i.e., different classes of combinatorial optimisation problems. This lead us to define the following hypothesis:

> Adapt and refine existing SBSE techniques, i.e. combinatorial optimisation methods, to an hybrid scheme of complete search tools with incomplete search approaches, inside an application driven solver, will efficiently reduce search and improve their exploration efficiency of most of SBSE problems, i.e. combinatorial optimisation problems.

Then, SMT solvers appears as key element for our hybridisation task: they belong to the complete search tools and also are application driven solvers. However, SMT solvers development lacks open and standard interface for modify its heuristics components without need of reassemble the whole tool. This problem has been partially covered in Z3 theorem prover, by the inclusion of a strategy language which allow users to exert control over the heuristic components to solve a SMT instance, i.e. a software engineering problem. But, to take full advantage of this language, expert guidelines for the construction of strategies are needed, and

most of end-users do not have the understanding or knowledge of all heuristics tools inside the solver. In this scenario local search techniques take importance, in order to automatically generate strategies which improve SMT solver performance. Thus, Z3 solver could be seen as a collaborative solver between complete or systematic search techniques and local search criteria. Once this scheme is defined, we define the following main and specific goals in order to answer our hypothesis.

### 1.1.1 Main goal

To fulfill our motivations, we define the following main goal:

*To investigate and study the impact of Hybrid Algorithms in Search-Based Software Engineering providing more generic and reusable solvers for a different classes of combinatorial optimisation problems.*

Note that as Software Engineering problems can be modelled as Search Problems, the solving advances achieved in Search-Based Software Engineering will help to improve the resolution procedures used in the field of Combinatorial Optimisation.

### 1.1.2 Specific Goals

In order to reach our main goal, we define some specific objectives for this thesis:

— Model and build a specialised framework to generate strategies for SMT solvers, in order to address different class of problems and extract good problem encoding from well-known models that will help to improve solving efficiency.

— Define an hybrid collaboration tool between a systematic approach and metaheuristics algorithms to improve the resolution of Search-Based Software Engineering problems, i.e. combinatorial optimisation problems.

— Use designed framework and tools, in an autonomous environment, for addressing algorithm building and/or selection to solve efficiently a selected combinatorial optimisation problem.

## 1.2 Research Contributions

The contributions of this research work are the following:

— Building a framework, based on a set of rules as intermediate layer, to apply several local search techniques without the need of modifying its structural components for addressing a selected combinatorial optimisation problem.

— Address the Strategy Challenge in SMT [10] through a framework for the automated generation of SMT strategies in an autonomous system environment.

— Address the SBSE "*Search for strategies rather than instances*" challenge [2], through the hybridisation of a systematic search system with local search algorithms.

— Introduce SMT [4] as useful systematic search system to integrate with others solving techniques in the Combinatorial Optimisation field, e.g, metaheuristics.

## 1.3 Overview of the Thesis

This thesis is divided in two parts: State of The Art and Contributions. The State of the Art part describes the foundations of this research, and the theoretical framework of the SMT solvers that will be considered, and it is covered by Chapter 2, 3 and 4. The Contributions part of this thesis shows an incremental work by using several engines in order to generate more efficient strategies to improve the Z3 performance. These engines are applyed by using a framework defined for the autonomous generation of strategies in Z3. Structure, analysis and results of these algorithms are presented from Chapter 5 to 11.

### 1.3.1 State of the Art

We begin in Chapter 2, where we revisit related works that serve as basis and motivation of this research. We build a path since the foundations of Combinatorial Optimisation; passing by Metaheuristics, Evolutionary Computing, and Parameter Control and Tuning; until specific trends as Autonomous Systems, Search-based Software Engineering and Automated Deduction.

In Chapter 3, we deepen in Satisfiability Modulo Theories (SMT). We analyse its immediate basis, the Boolean Satisfiability problem (SAT), its applications in real-life systems, and how they are addressed to be solved. Also, we introduce *The Strategy Challenge in SMT*, as essential milestone of this research, and how we will address it in order to improve SMT solvers performance.

Finally, in Chapter 4, we define and analyse what is a *Strategy* in SMT, focusing in the Z3 solver. We formalise a term grammar which could derive well-formed strategies, and check how these strategies help to improve Z3 performance.

### 1.3.2 Contributions

It starts in Chapter 5, where we formalise a framework for automated generation of strategies in Z3. This system is composed of two core elements, a rule approach and an engine system. The former defines how to modify strategies using rules based in evolutionary computing op-

erators, the later defines how algorithms, or engines, apply these rules and the configuration of this procedures.

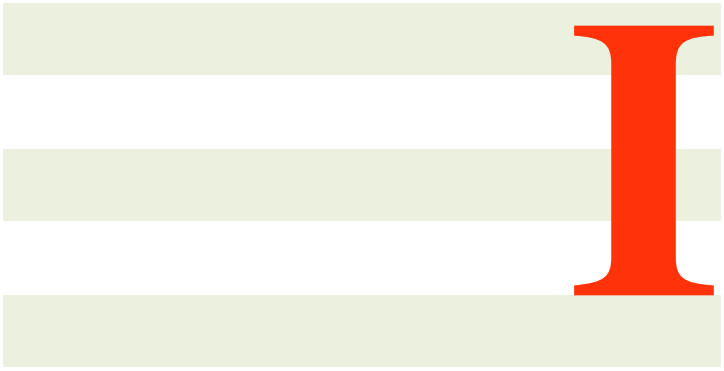
In Chapter 6, we present two simple approaches to generate optimised strategies by using different types of rules separately. The goal is to decide which kind of rule is more relevant for this task. The algorithms used are a evolutionary parameter tuner for strategies, called StratTUNE, and a evolutionary strategy generator, named StratGEN.

In Chapter 7, a Tree-based Grammatical Genetic Programming Algorithm, called StratEVO, is defined and implemented. This algorithm uses the term grammar defined in the State of Art as base for modeling strategies as trees to ease the application of modification rules to generate strategies. StratEVO is capable of generate complex efficient strategies for several SMT logics by only using the more relevant type of rule with respect to the performance determined in Chapter 6.

We revisit StratEVO, in Chapter 8, to generate schemes for collaboration between different types of rules. We use different off-line and on-line hybridisation approaches to improve StratEVO performance.

Last application step, Chapter 9, introduces a new set of rules to handle the semantical component of time configuration. Here, algorithms includes rules to spread the distribution of time given in the learning process to scenarios of execution where the time-out per instance are up three magnitude order larger.

Finally, in Chapter 10, we analyse the contributions of this works, we discuss perspective steps to take in the future, and how this system could be extrapolated to improve performance in several related computer science fields.



# State of the Art

# 2

# Towards an Autonomous System

In the following chapter, we revisit the advances achieved through the development of Informatics areas that serve as foundations of this work.

## 2.1 Combinatorial Optimisation

Since its historical development in Economics, as stated in [1], the field of Combinatorial Optimisation has changed the trivial-look over discrete optimisation problems and the finite domains of its decision variables: *Today, we see these problems abound everywhere*. Their current applications follows the same path: *they are finite but they seem endless*. This inherent combinatorial-space large-size feature, sometimes mixed with a complex evaluation function and/or a hard set of constraints, made intractable the task of finding an optimal solution.

The mentioned behaviour in combinatorial optimisation problems is not just an empirical conclusion, it is also presented as the NP-completeness framework of computational complexity theory [5, 11, 12], which implies that even if a solution for a hard problem could be easily verifiable in polynomial time through a deterministic Turing machine (that is to say, at least in the NP class), the procedure of founding a solution is not trivial in function of computational and time resources. As it is not our goal to discuss in depth the concepts and proofs of computational complexity, we refer the reader to well-known books as [13, 14, 15].

To address this kind of problems, we should work in a reduced subset of problem instances (an easy solvable selection), or rely on procedures based in higher level strategies and step-by-step heuristics improvements to generate approximated optimal solutions. These approx-

imation algorithms or **Metaheuristics**, have been preferred to solve combinatorial problems, despite not being able to certify the optimality of the solutions found as exact procedures. Nevertheless, these methods have been proved as incapable to match solution quality obtained by metaheuristics, particularly for real-world problems, which often attain notably high levels of complexity [16].

### 2.1.1 Combinatorial Optimisation Problems

As stated in [17], and following the black-box method of computer systems, a problem could be defined as a *input* transformed into a *output* by means of a transformation *model* (see Figure 2.1).

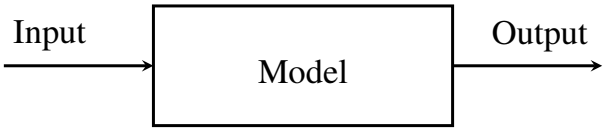



Figure 2.1 – Problem definition as a black-box method.

This concept leads to classify problems as follows:

1. Optimisation problems: Given a known model and an expected or specified output, the task is to find an input configuration. Note that, the output is not necessarily known, but could be implicitly defined, e.g. find the minimum value of a distance function. Thus, instead specifying the exact value of the output, we expect that its value is smaller than all others by finding inputs for realising this.
2. Modelling problems: Given some known inputs and outputs, the job is to seek a model function that delivers the correct output for each known input, i.e., to find a model that matches previous experiences, and can hopefully generalise even unseen instances. Note that modelling problems can be transformed into optimisation problems, by minimizing the error rate or maximizing its success rate of a proposed model used as input.
3. Simulation problems: Given the system model and some inputs, the task is to compute the outputs corresponding to these inputs. This type of problems are commonly used for prediction and forecasting jobs aiming to look into the future.

Combinatorial optimisation problems are related to the first two classes of problems, which have several candidates to solve the problem as inputs or models. The candidates for solving these kind of problems are located in a space of possibilities which is usually enormous. Then, the problem solving process is a *search* through a potentially huge set of possibilities to find a solution. Thus, combinatorial optimisation problems can be seen as search problems. This

leads to the definition of **search** concept: *to found, evaluate and accept solutions as candidates to solve a problem from the set of all possible solutions*. This set of possible solutions is called *search space*, which is generally defined by the domains of all *decision variables* that define a problem solution.

The evaluation of solution is performed using a specific function called *evaluation or objective function* that quantifies and compares candidates. The acceptance of the solution depends if a *set of constraints* is fulfilled. Then a candidate is a *feasible solution* if can be evaluated by the objective function and satisfies the requirements of the problem. Thus, search problems are also classified according to the components that evaluate and accept solutions, i.e., guide the search, as shown in Table 2.1.

| Constraints | Objective Function | |
|---|---|---|
| | Yes | No |
| Yes | Constraint Satisfaction and Optimisation Problem (CSOP) | Constraint Satisfaction Problem (CSP) |
| No | Free Optimisation Problem (FOP) | No Problem |

Table 2.1 – Search problems classification according to its elements.

If only a evaluation function is present, we face a *free optimisation problem*. Its counterpart are the *constrained satisfaction problems* which have to satisfy only a set of constraints. When both elements are present, we talk about a *constraint satisfaction and optimisation problem*. Commonly, a problem from a specific type can be mapped as another problem class, i.e., all these classification can used to address a same of problems. For illustrate this, we define the Example 2.1.

**Example 2.1** The famous constraint satisfaction problem, N-Queens problems [18], states the following:

*Place $n$ queens on a $n \times n$ chess board*
*in such a way that no two queens check each other.*

In Figure 2.2, an example with $n = 8$ queens is shown. This problem could be addressed as:

— Free Optimisation Problem: Given a search space $S$ of all board configurations with $n$ queens, we define an objective function $f$ that outputs the number of free queens for a given configuration. Thus, a problem solution is any configuration $s \in S$ with $f(s) = n$.

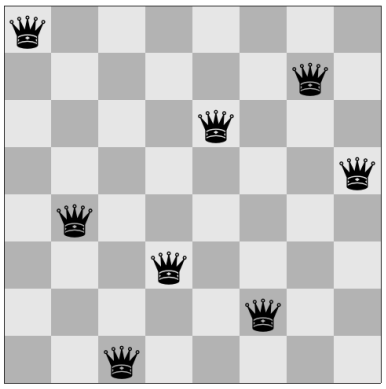


Figure 2.2 – N-Queens problem with $n = 8$ queens.

— Constraint Satisfaction Problem: Using the same search space $S$, we define a constraint $C$ that is satisfied only if and only if two queens are not checked. Then, a specific configuration $s \in S$ is a solution if satisfies $C(s) = \texttt{True}$.

— Constraint Satisfaction and Optimisation Problem: Using the same search space $S$, we define an constraint $D$ which is satisfied only if and only if no queens in the board are in the same column and row, and a evaluation function $g$ counting the amount of diagonal checks between all placed queen. The solution of this problem is a configuration $s \in S$ such that $g(s) = 0$ and $D(s) = \texttt{True}$. ♢

### 2.1.2 Metaheuristics

Before define the concept of metaheuristics, and its more relevant procedures for this thesis, let us remark the two existent classifications of search paradigms stated in [19]:

— Local Search vs Systematic Approximation: Systematic Approximation look for *completeness*: traverse the search space in a systematic way which guarantees with certainty either if a optimal solution is found, or if no solution exists. Local search points to move partially, called *incompleteness*, in the search space from a known starting point, moving through different small sub-spaces of the search space. Differently from its systematic counterpart, local search cannot guarantee the existence of a feasible or optimal solution, and the fact that no solution exists can never be determined with certainty.

— Constructive vs Perturbative Search [20]: Constructive search aims to build a good-enough feasible solution from scratch, generally extending incomplete candidate solutions. Perturbative Search looks to modify one or more elements of a existing candidate solution in order to generate improvements.

Therefore, we can understand a metaheuristic as an **automatic process to search a "good-enough" solution**, either building or repairing, with regards to a finite, but intractable, set of

candidates. Note that, both objective function and constraints bound the search space in terms of feasible solutions. Nevertheless, unfeasible solutions can be a bridge between disjoint feasible-portions of the search space, then soft some constraints to be unsatisfied, and quantify them in the evaluation function for fair comparison, can help to improve metaheuristics performance.

#### 2.1.2.1 Local Search

Local search procedures are mostly based on repairative or perturbative search. They rely on search improvements inside the local scope of a candidate solution, or *neighborhood*. The neighborhood is determined by applying *movement* in the current candidate: a perturbation in eligible solution components, based in local knowledge or *heuristic*.

Originally, local search techniques were designed for improve a solution in a single-objective scope. The intuitive iterative improvement, Hill-Climbing, serves a starting point for more robust techniques as Simulated Annealing [21], Tabu Search [22], GRASP [23] or Generalised Hill Climbing [24].

As everyday local search techniques turn more sophisticated, the key simplicity and generality of them have been push-aside in search of better performance, producing confusion between heuristics and metaheuristics. Iterated Local Search [25] arise to clearly define between the general procedure strategy (metaheuristic) and the decision-making based in problem-specific knowledge (heuristic). Also, local search techniques have been adapted and extend to gain a huge space in multi-objective optimisation, as well summarised in [26]. Also, their use allowed to define multi-objective standalone local search algorithms as Pareto Local Search [27, 28] or Bi-criteria Local Search [29].

For more detailed information in the mentioned techniques, algorithms adaptations to others scopes, we refer the reader to Hoos and Stützle Stochastic Local Search [19] compilation and applications.

Notwithstanding, one of the most interesting applications are on the fields of parameter control and tuning, or *Parameter Configuration*, which will be analysed in Section 2.1.4, where algorithms are adapted as better as possible to a specific situation.

#### 2.1.2.2 Evolutionary Computing

The set of techniques based on the principles of Darwin's theory on evolution and natural selection [30] are circumscribed in the *Evolutionary Computing* or Computation (EC) [17] area. Despite being novel concept (1990s') its main techniques were separately developed for almost two decades before. These techniques or *Evolutionary algorithms* (EAs) [31] are characterised by working over a set of *individuals* (candidate solutions) called *population*, rather than working over a single candidate as in Local Search. Then, EAs modify population individuals via

evolutionary or biological based-operators as selection, mutation, recombination (or crossover), etc. Four major trends are considered:

1. Evolutionary Programming (EP) [32, 33, 34]: Developed by Fogel et al. in the 1960s as simulation of evolution. EP aims to evolve different species, represented for a real-value vector of features, in a population driven by a mutation operator.
2. Evolution Strategies (ES) [35, 36]: Invented in Germany also in the 1960s and developed in the 1970s by Rechenberg and Schwefel. ES creates an offspring set from the original population, where individuals are real-value vectors, through operators as mutation and recombination. Then, the best $n$-ranked individuals of the union-set of parents and children make up the next generation, if *plus scheme* $(\mu + \lambda)$ is selected; or the parent population is completely replaced by the offsprings generated, if *comma scheme* $(\mu, \lambda)$ is chosen.
3. Genetic Algorithm (GA) [37, 38, 39]: Initially proposed by Holland in 1973 and rectified by De Jong in 1975. GA evolves a population, where an individual is represented by a binary vector, using a combination of crossover, mutation and selection, the former being the predominant operator. These operators, with exception of selection, have a defined occurrence probability, therefore they are not always applied.
4. Genetic Programming (GP) [40, 41, 42, 43]: Born in the 1990s, and run by John Koza, Genetic Programming is a particular application of Genetic Algorithms, which is powerful enough to be considered as a stand-alone Evolutionary Computation trend. It works similar as GA, but individual representation is made using tree structures, which generally represent models, syntax or grammars of mathematical expressions or computer programs. Therefore, all operator are defined to work over a tree structure.

#### 2.1.2.2.1 Grammar-based Genetic Programming

Despite being the youngest trend in Evolutionary Computation, Genetic Programming has served as starting point for techniques that includes its principles. The most interesting advances start from the use of formal grammars in order to model individuals and its evolution process, called *Grammar-based Genetic Programming* (GbGP) [44]. The advantages of using grammars include assure the closure property, i.e., any individual generated by genetic operators must be evaluable, and to bias the GP structures, i.e., individuals typing and syntax are easily maintained by manipulating the explicit derivation tree from the grammar, as explained by Whigham [45] in one of the first works known in the area.

Actually, one of the most used variations is *Grammatical Evolution* (GE) [46, 47]. GE uses a grammar to map a integer-value vector into a code segment thanks to a simple enumeration of the production rules, as shown in Example 2.2.

**Example 2.2** Let $G$ be a grammar representing basic arithmetic operations, addition ($+$) and multiplication ($\times$), between two expressions composed by two variables, $X$ and $Y$, by using the following production rules:

$$\begin{array}{llcll} \text{A)} & \texttt{exp} & \rightarrow & \texttt{exp op exp} & (0) \\ & & | & \texttt{var} & (1) \\ \text{B)} & \texttt{var} & \rightarrow & x & (0) \\ & & | & y & (1) \\ \text{C)} & \texttt{op} & \rightarrow & + & (0) \\ & & | & \times & (1) \end{array}$$

The basic operation $x \times y$, can be mapped as the following array of integers:

$$I = \begin{array}{|c|c|c|c|c|c|} \hline 24 & 307 & 78 & 155 & 229 & 43 \\ \hline \end{array}$$

Starting in the symbol `exp` and reading $I$, from left to right, we could generate the mentioned expression by using the following scheme for rule selection:

$$\text{Selected Rule} = \text{Array Component Value } \% \text{ Number of Rules for Symbol}$$

where $\%$ is the classic arithmetic modulo operator. Thus, the following table, shows how $I$ is decoded.

| Expression | Current Symbol | Current Array Value | Total Rules | Selected Rule | New Expression |
|---|---|---|---|---|---|
| `exp` | `exp` | 24 | 2 | $24 \,\%\, 2 = 0$ | `exp op exp` |
| `exp op exp` | `exp` | 307 | 2 | $307 \,\%\, 2 = 1$ | `var op exp` |
| `var op exp` | `var` | 78 | 2 | $78 \,\%\, 2 = 0$ | $X$ `op exp` |
| $X$ `op exp` | `op` | 155 | 2 | $155 \,\%\, 2 = 1$ | $X\times$ `exp` |
| $X\times$ `exp` | `exp` | 229 | 2 | $229 \,\%\, 2 = 1$ | $X\times$ `var` |
| $X\times$ `var` | `var` | 43 | 2 | $43 \,\%\, 2 = 1$ | $X \times Y$ |

Note, for each step the chosen symbol is the one located to the left on the current expression. ♢

This approach holds the properties mentioned above and adds: use any type of language to be evolved, handle a population with individuals of different size, and avoid the inclusion of several introns while mutating the population.

#### 2.1.2.2.2 Semantic-based Genetic Programming

However, evolving mathematical expressions or code programs, regardless the use of grammars, is done purely over a syntactic space [48]. Syntax-driven search could lead to *meaningless* efforts, because some traded expressions could be radically different but generate no impact at all, therefore this "meaning" also constraints the evolution process. The meaning of the syntax symbols or expression over a defined context is what we call *Semantics* (as in linguistics). Since 2012, the use of semantics in Genetic Programming to avoid blind-search over the meaning of the generated candidate solutions have increased, but still unexploited. We will refer this as *Semantic-based Genetic Programming*.

Semantics have several definitions, as stated in [48, 49], but could be reduced to one common element: *fitness*, the function which encode the genotype as phenotype. Thus, the evaluation function generates the semantics values of the set of solutions, called *Semantic Space*. In this new space, as shown in Example 2.3, we could found that very different solutions in structure (genotype) are semantically equivalent (phenotype), or small configuration differences lead to great evaluation variance.

**Example 2.3** Let $S$ be the set of all boards configuration for N-Queens problem (shown in Example 2.1). Let $f$ be a fitness or objective function which evaluates how many queens are checked for a given board configuration $s \in S$. Figure 2.3 shows how different boards configuration are mapped to different regions of the Semantic Space by means of the fitness

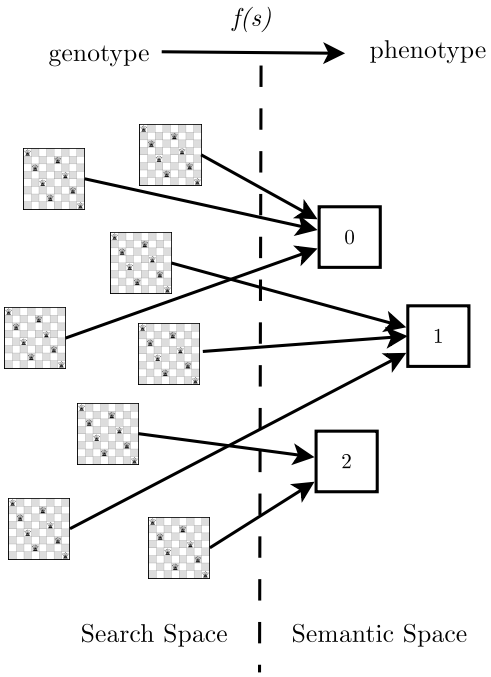



Figure 2.3 – N-queens Semantic Space example: several boards configuration mapped into Semantic Space by means of an objective function $f$.

or objective function $f$.

*Two boards configuration, $s_1, s_2 \in S$ with $s_1 \neq s_2$, are semantically equivalent if and only if $f(s_1) = f(s_2)$.*

Lets us remark, that several boards configuration could be mapped to the same Semantic Space region. ♢

Classification to the inclusion of semantics in Genetic Programming has been done in [48, 50], but we rather classify the most important advances as follows:

1. Semantic-constrained grammars: In order to include semantical information to the candidate solution generation, formal grammars with a set of production rules based on their meaning are used. Then, individuals are validated by a parse tree including some semantical constraints attached to the original grammar. Some relevant works includes the use of Attribute grammars [51, 52, 53], Christiansen grammars [54, 55] and Logic grammars [56, 57, 58].

2. Semantic-guided operators: Several operator have been created to add semantics elements to candidates solutions. Semantic-driven operators [59, 60] works as regular Genetic Programming operator, but avoiding create semantically equivalent solutions. Semantic distance concept, distance between two solutions in the semantic space, have been simultaneously defined, forked and used in Nguyen's semantic operators [50], Locally Geometric Semantic operators [61] and Approximate Geometric Semantic operators [62]. The latter is the root point of the very promising Geometric Semantic Genetic Programming [49, 63], the first kind of algorithm which applies genetic operators directly in the semantic space, but with a recoil of solution size overgrowing. Others operator uses structure information for semantic reduction, as in [59, 60, 64], which removes redundant and isomorphic substructures in binary trees.

3. Semantic-guided population: Population diversity in Genetic Programming is the key to avoid premature local optima blockage [65, 66, 67], this have been taken to generate semantic-diverse population, especially in population initialisation [68, 69], in order of a better exploration of the semantic space. Also, some semantic-guided operator aims for diversity in the evolving process as in Semantic-driven operators [59, 60] or Nguyen's Semantic-aware operators [50].

4. Formal methods: Retrieving systems internal information through mathematical techniques, used for specification, development and verification of software and hardware systems [70], in order to infer semantical information. Appraised work includes the use of Abstract interpretation [71, 72] and Model Checking [73, 74] procedures.

Figure 2.4 condenses the classification of the explained Evolutionary Computation categories relevant for this research.

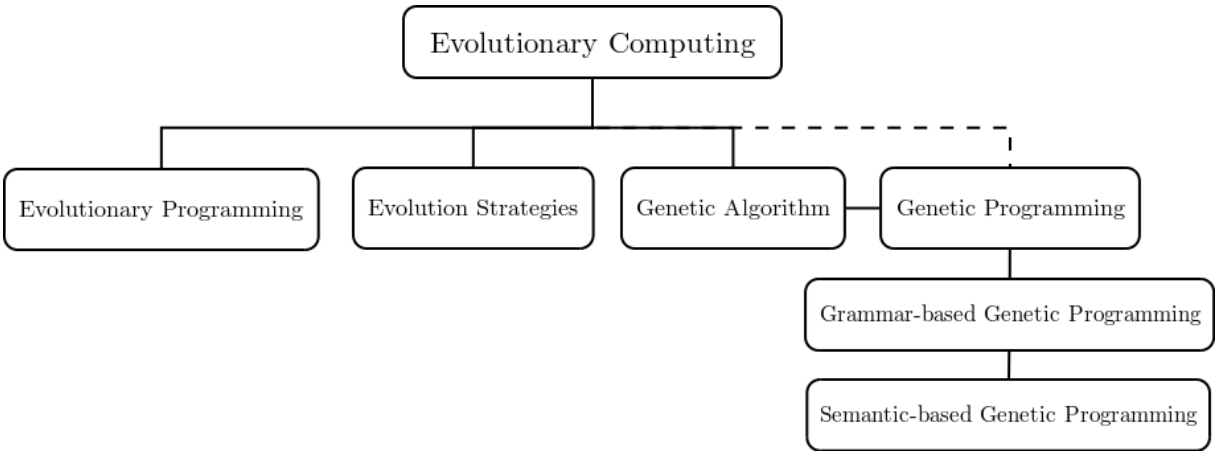



Figure 2.4 – Evolutionary Computing: techniques classification

### 2.1.3 Autonomous Search

The success of approximation algorithms relies on the ability to fit to a specific problem, in order to reduce the gap with the unknown global optimum. Two elements are identified as key in this process:

1. Be suitable, i.e., be an efficient implementation option to solve the problem, and
2. Have a correct local configuration, i.e., use the best algorithm setting values to improve performance.

These features have been largely studied as part of *Algorithm Selection Problem* [75], and used as starting point for *Autonomous Search* [76] systems. The goal of these systems is to *provide an easy-to-use interface for end users, who could provide a possibly informal description of their problem and obtain solutions with minimum interaction and technical knowledge* [77]. This is achieved through the use of several Combinatorial Optimisation problem solving ideas, algorithms or techniques, which serve as partial solution in an ideal non-human assisted system, e.g. hyper-heuristics, portfolio optimisation, evolutionary computing, adaptive or reactive methods, and so on.

An ideal Autonomous Search system will use the most suitable solving technique and its best available configuration in order to solve a problem, based on the existing and generated knowledge, always inside the real-life solving constraints as computing resources or time consumption, i.e., **automatically**, as shown in Figure 2.5.

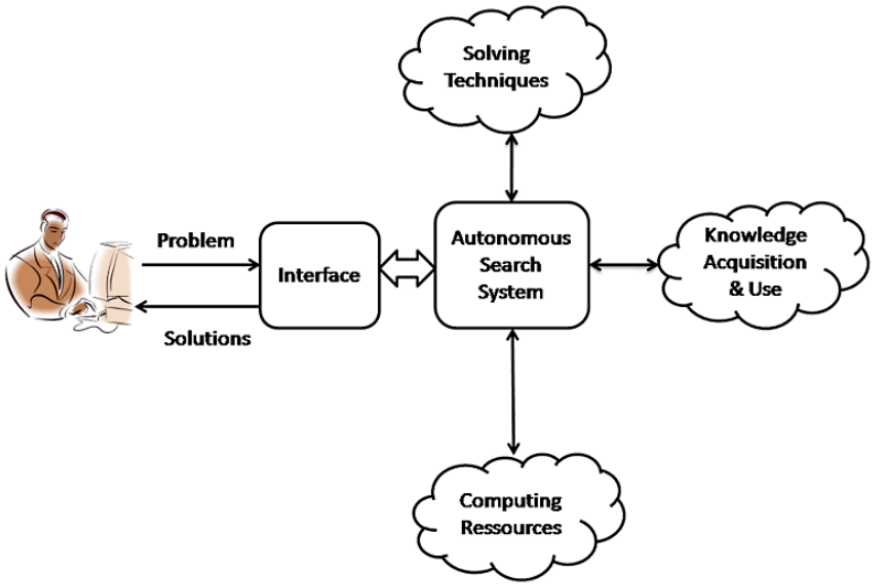



Figure 2.5 – Autonomous Search System: ideal interaction scheme [76].

In this thesis, we focus in two kinds of metaheuristics which are generally used in Autonomous Search systems: *Local Search* and *Evolutionary Computing*. These metaheuristics are used in one of its most critical applications: parameter control and tuning, which we call *Parameter Configuration*.

### 2.1.4 Parameter Configuration

Most metaheuristics rely on their local settings to fit as better as possible to the problem and to obtain good performances. Despite of being successful, metaheuristics are generally used with configuration values defined by conventions, ad-hoc choices and/or empirical conclusion in a limited scale [78] (e.g. three cases over a thousand). Therefore, is hard to know if the chosen settings really fit the algorithm and, given their impact, optimise its performance for a defined scope [79].

*What configuration fits better?* This is a task that not all researchers or developers are aware of, and little effort is spent with regards to their potential effects. Also it is a non-trivial problem by the following reasons:

1. Time consuming: it depends on several execution of the metaheuristic.
2. Problem related: best/better configurations change depending on the problem.
3. Interelated: parameters are generally related by complex and/or non-linear interactions.

We could address parameter configuration task by two approaches, defined in [80] and shown in Figure 2.6: Parameter Tuning and Parameter Control.

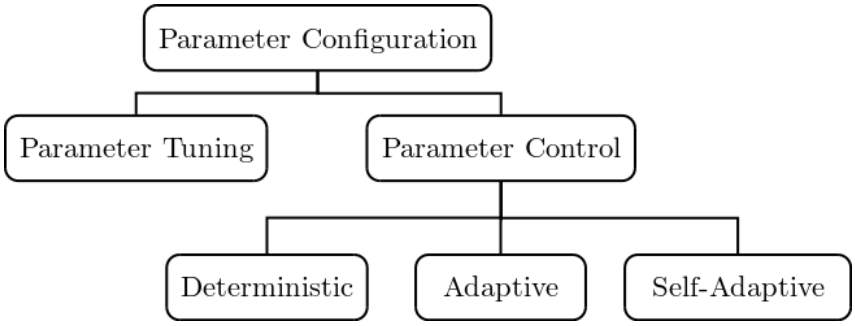



Figure 2.6 – Parameter configuration classification

Both schemes search for an common goal: to obtain an **automatic algorithm configuration** in order to maximise its performance. This is a high-impact topic in Evolutionary Computing [81] and Autonomous Systems [76].

#### 2.1.4.1 Parameter Tuning

Parameter Tuning advocates to change algorithm configuration in an **off-line procedure**, i.e., the setting values change outside the algorithm execution. Therefore, an interative execution process is the path to evaluate how good is a configuration w.r.t others settings. As stated in [82], several ways exist to tune parameters:

— Arbitrary criteria: Use and modify parameter configuration based in designers/users own-criteria.

— Analogous criteria: Follow established guidelines, used and defined in recognised works, to set parameter values.

— Experimental criteria: Generate a sample through experimental design to decide or define by statistical tools an appropriate setting.

— Search criteria: Use search methods to define a good algorithm setting from the space of all available combinations, i.e., use metaheuristics to generate a good parameter configuration.

— Hybrid criteria: Combination of above methods, generally mixing Experimental and Search criteria.

Several tuning algorithms exists, which have specially a great impact for Evolutionary Algorithms [83] given the amount of parameters present in them. The most relevant techniques in the literature, regarding this work, are summarised as follows:

— F-Race [84, 85, 86]: Based on racing procedures, F-Race is an iterative procedure which evaluates several parameter configuration through different benchmarks instances. At each step, candidates configurations or *racers* are contrasted through a non-parametric

statistical test, Ranked-based Friedman [87], to disqualify the ones who are far-away from the leader. This discrimination is done to reduce computational resources wasted in poor performance configuration and focus them in more promising options. But, F-Race is impractical to use when the amount of parameter is large, because it has to evaluate all combinatorial as initial state, turning into an intractable process. Sampling F-Race and Iterative F-Race, proposed both in [86], tackle this problem. Sampling F-Race use a subset, sampled from the space of all configurations, as initial state to then apply F-Race. The success of Sampling F-race depends on the subset size [79]. Iterative F-Race, improves Sampling F-Race making it an two-step iterative process. Thus, the first step of the iteration is to choose a sample from a known probabilistic model, and the second step is to apply F-Race. The winner and runners-up of a race serve to update the mentioned model in the next iteration.

— ParamILS [88, 89]: A Iterated Local Search algorithm framework designed for parameter tuning, being one of the most used tools for algorithm configuration. ParamILS repairs an algorithm configuration by performing a local search procedure designed to quickly reach or approach a locally optimal setting. Then, this new candidate is perturbed to escape from local optima. Best candidate between iterations is saved by a defined criteria, to then reset the process from the last checkpoint: better overall solution found. The criteria defined to decide that one setting is better than other are: BasicILS and FocusedILS. The former compares two configuration performance in a subset of fixed size of all benchmark set. The latter uses dominion concept to block already compared configuration which are worst than others, thus unnecessary execution on poor-performance configuration are avoided.

— CALIBRA [90]: A hybrid criteria method combining experimental design and local search. It samples the configuration space to found promising settings areas to then uses local search to improve these solution by the use of Taguchi Orthogonal Arrays. Regrettably, it usually works with tiny amounts of parameters.

— REVAC [91]: Relevance Estimation and Value Calibration is a evolutionary algorithm for parameter tuning which includes sampling, but in this case with regards to the value distribution of each parameter and not the candidate distribution in configuration space. The distribution estimation is done to initialise the population in order to diversify over the search space. Then, through specially designed operators for crossover and mutation, in each generation the range of possible values of parameters is reduced to focus the search. Pitifully, REVAC needs of several parameters to work.

— EVOCA [92]: The Evolutionary Calibrator was born in order to improve REVAC flaws as lose relevant parameter information when reducing the set of possible values to choose, or the introduction of new parameters to make the algorithm work. EVOCA evolves a

population generated from an uniform sampling of the variable dominion, also is this element which decides the population size. Also, crossover and mutation elements works stochastically reducing the amount operator parameters, and they allow introduce values that are forbidden in REVAC because of premature locality.

For more detailed information about tuning methods, their classification and scope we refer the reader to [82, 83].

#### 2.1.4.2 Parameter Control

Parameter Control scheme allows change the parameters values in an **on-line procedure**, i.e., transform the values during the execution of the metaheuristic. The goal is to use different parameter settings through an algorithm run to face as better as possible its different steps, because a fixed parameter setting *does not guarantee optimal algorithm behaviour, since different algorithm configurations may be optimal at different stages of the optimisation process* [93]. As stated by Eiben in [80], Parameter Control procedures could be classified in:

1. Deterministic: Parameter values are controlled using a deterministic rules applied, i.e., without assistance of the search information. Generally, a time schedule is used to activate these rules.
2. Adaptive: Algorithm setting is modified using feedback from the search process in order to determine the strength and/or direction of the changing process.
3. Self-adaptive: Parameters are encoded as part of candidate solutions, then metaheuristics can modify their own parameter values during execution through their search method. This is known as co-evolution in Evolutionary Algorithms.

Our concern is in Adaptive Parameter Control which has several involvements in the use of metaheuristics.

In Local Search, the biggest trend is Reactive Search Optimisation [94, 95, 96], led by Battiti and Brunato. It integrates symbolic notions of machine learning on behalf adaptive behaviour. Some examples are adaptive versions of classic local search metaheuristics as: Reactive Tabu Search [97, 98, 99], Adaptive Simulated Annealing [100, 101, 102] and Adaptive Random Search [103]. Lately, Reinforcement Learning have been studied to be successfully inserted in these kind of techniques. [104]

In Evolutionary Algorithms, parameter control has not been exploited as parameter tuning, despite the former can solve the problem addressed by the later [105]. Several promising works have been achieved, but they do not integrate all elements of standard EAs. As mentioned in [105], most of the focus is given to parameters controlling population and variation operators (as crossover and mutation). Also, most adaptive techniques are phenotypic [93], i.e., changes in parameters values respond to the information given by the evaluation function

at some execution stage, leaving in background some other relevant information. The diverse work done for controlling population size and behaviour are well summarised in [106, 107], meanwhile for variation operators most recurrent works are about Adaptive Operator Selection [108, 109, 110, 111], a technique which *autonomously select which operator, among the available ones, should be applied by the EA at a given instant, based on how they have performed in the search so far*.

For more detailed classification of techniques used for controlling parameters and their trends, we recommend [93, 105] surveys.

## 2.2 Search-Based Software Engineering

As explained before, combinatorial optimisation problems, i.e., search problems, tends to appear everywhere, even in several non-related areas. In Computer Science, Software Engineering tasks were typically seen as more practical and human-experience dependent [2], i.e., software engineer knowledge played an unique fundamental role in order to solve well-known complex problems. Despite this, the inclusion of search algorithms in order to automate software engineering problem resolution and support the decision making in software life-cycle problems have been arise and settled as a common practice in the last decade. This turned into one of the biggest applications of search algorithms to the date.

In 2001, Mark Harman et al. [112] grounded the first glimpses of this promising area as *Search-Based Software Engineering* (SBSE). SBSE promotes the use of metaheuristics to solve well-known software engineering problems, which could be mapped as search problems. Software Engineering now is one of the biggest application of combinatorial optimisation methods.

Search-based Software Engineering also introduced some interesting points of comparison between software common elements and evolutionary algorithms structure, e.g. software metrics with fitness function, bridging evolutionary computing to the field. As consequence, Software Engineering field opened their doors for several related Artificial Intelligence (AI) as Predictive Modelling [113, 114], Clustering [115], Machine Learning [116] and Automatic Deduction [117]. Moreover, in [2] some interesting challenges were defined to address the use of AI techniques for software engineering problems, being the most relevant the goal of *searching for strategies rather than instances*, aiming to solve class of problems rather than specific problem instances. This is well-known objective in optimisation problem solving, being a foundamental basis of the **Autonomous Search systems**, as explained in Section 2.1.3.

As Harman stated in [118, 119], Testing problems [120, 121] have taken most benefit from SBSE. Despite this, search procedures have gained quite interest in other core Software Engineering topics as Project Management [122], Maintenance [123, 124], Design and Architecture [125, 126], Program Analysis [127] and Software Improvement [128, 129, 130].

### 2.2.1 Genetic Improvement

One of the biggest features of software is its capacity of evolve [131]. Despite not being formulated as a technical feature, it responds to adaptive capacity of software to fulfill the changing requirements and environments in which they operate. This evolutionary software property have served as one of the main motivations of using Evolutionary Algorithms in Software Engineering problems. As mentioned in [129, 132], Evolutionary Algorithms have been used around 71% of all research done in SBSE until 2011, being Genetic Algorithms and Genetic Programming the most used techniques.

The intrinsic capability of Genetic Programming to evolve code, explained in Section 2.1.2.2, has been the starting point of *Genetic Improvement* (GI), i.e., automated search to find improved versions of existing software or software improvement. GI has triggered huge improvements for a diverse set of software performance properties: execution time, energy and memory consumption; as well as repairing and extending software system functionality [129]. Some examples are Software Transplantation [130], where some portion of code from one system is exported to another, entirely unrelated, system; or Software reengineering [128], where a genetic programming algorithm evolves a software to perform faster. Nevertheless, several SBSE areas inside Software Improvement are not yet exploited, having a great potential: Software Synthesis, Repair and **Transformation**, **Parameter Tuning**, etc. More details of GI could be found in [129] survey.

## 2.3 Automated Deduction

Automated Deduction or *Automated Theorem Proving* (ATP) is the task of given a formula, try to automatically evaluate if it is universally valid or not, through the uses of logic-based computational programs [117]. Algorithms for automated deduction were developed well before computational tools building or software engineering became a field, using first-order logic which is rooted in logic and foundations for mathematics. First-order satisfiability is concerned whether there is an interpretation (model) that satisfies a given formula.

Despite being initially far from combinatorial optimisation scope, huge improvements of theorem proving tools came with the boolean first-order logic or, *Propositional Logic*, and the search methods to address the most famous Constraint Satisfaction Problem, Boolean Satisfaction Problem [5] or SAT.

Moreover, Search-based Software Engineering problems, i.e., search problems, have used automated deduction solvers as one of the most relevant tools for solving different classes of problems based on their natural connection: *logics can be an intermediate layer between software problems and automated deduction tools, problems can be mapped as queries in a logical satisfiability space, and tools could focus to solve them. The connection between logic and*

*software was established early since its foundations, because before a programming language or software existed, logical formalisms for calculus, automata, and equational reasoning were building the basis of their development* [3].

Note that, each software is computation piece of instructions following a defined logic, therefore a logic can be used to characterise all the possible events of a software piece, either inputs to a program, values assignation to variables at some execution point, the possible steps to follow in a program, the effects of these execution steps on the program state, or the properties of execution traces of a program. Thus, first-order satisfiability has been very useful specially for software validation or verification problems [117].

### 2.3.1 Boolean Satisfiability Problem

Propositional logic is directly related to SAT [5, 133, 134] formulas and could be seen as the smallest sensible subset of first-order logic [3]. SAT formulas are checked efficiently by modern SAT solvers. This allowed to SAT-solving turns into a highly active research area. The satisfiability problem is Constraint Satisfaction problem and it is simpler to represent: a model for a propositional formula is an assignment of the propositional variables to truth values 0 (false) or 1 (true), such that the formula evaluates to 1 (true) under this assignment.

With respect to Search-based Software Engineering, SAT-solving has been applied to software verification through model checking [135, 136, 137], software analysis [138] and software testing [139]. However, binary nature of SAT formula have been a major barrier in order to model several search problems into SAT domain.

### 2.3.2 Satisfiability Modulo Theories

The theory of arithmetic has been a central topic in logic ever since symbolic logic took shape [3]. Satisfiability Modulo Theories (SMT) [4] has been risen generalisation of SAT which includes support for domains that are commonly found in programs (integer, real, linear and non-linear arithmetics, arrays, bit-vectors, pseudo-booleans, etc) and specialised algorithms to solve these type of mathematical assertions. This contrasts pure first-order theorem proving that has no built-in support for those domains, which are critical for modeling and analyzing software systems.

As shown in [3, 4, 9], several applications for SMT solvers exist in Software Engineering, different from classic Program Verification problems [140, 141], including: Symbolic Execution Testing, Program Analysis, Static Runtime Checking, Test Case Generation and Software Modeling.

More detailed insights about SMT could be found in his dedicated analysis in Chapter 3.

## 2.4 Conclusions

In this chapter, we have presented the main foundations of this work, classic Combinatorial Optimisation procedures for Search Problems, and how are they used for Autonomous Search, the idea of automatically solving a problem with minimum end-user interaction.

Also, we review important topics as Metaheuristics and how their are used in autonomous procedures mainly focused on Parameter Configuration.

Moreover, we have introduced a current application for these techniques, Search-Based Software Engineering, and how their own challenges could be addressed from search related methods, as Evolutionary Computing and Automatic Deduction: the former being essential part of Metaheuristics, and the later includes the popular combinatorial problem: SAT.

# 3

# Satisfiability Modulo Theories

In this chapter, we introduce Satisfiability Modulo Theories (SMT) concepts, scopes and applications. Also, we explain the importance of SMT solving tools and we address one of the most important challenges in the field. Before describing SMT, Constraint Satisfaction problems and Boolean Satisfiability problems (SAT) should be explained.

## 3.1 Boolean Satisfiability problem: SAT

Boolean Satisfiability problem, called SAT [133, 134], is a fundamental problem in Computer Science. Given a set of propositional logic formulas ($F = \{f_1, f_2, \ldots, f_n\}$) over boolean variables ($x_1, x_2, \ldots, x_n$) related by logical connectives (NOT $\neg$, AND $\wedge$, OR $\vee$), SAT problem aims to decide if they can be evaluated as *true* by choosing *true* or *false* values for its variables. It is famous for being the first demonstrated NP-Complete problem by Cook-Levin [5] theorem in 1971.

**Example 3.1** Let $x_1, x_2, x_3, x_4$ boolean variables, defining the formula set $F = \{f_1, f_2, f_3, f_4\}$, with $f_1 = (\neg x_1 \vee x_2)$, $f_2 = (\neg x_1 \vee x_2 \vee \neg x_3)$, $f_3 = (\neg x_1 \vee x_3 \vee \neg x_4)$ and $f_4 = (x_1 \vee x_4)$. What values must be assigned to variables in order to $\bigwedge_{i=1}^{4} f_i$ be true? $\diamondsuit$

In Example 3.1, as in most SAT instances and applications, formulas are presented in *Conjunctive Normal Form* (CNF). In CNF formulas, the *literals* (i.e., variables and their negations (NOT $\neg$)), are joined by disjunctive connector (OR $\vee$), building a formula or *clause*. Then,

clauses are joined by the conjunctive operator (AND $\wedge$). Also, this example has at most 3-literals per clause, called 3-SAT, which is one of the Karp's 21 NP-Complete Problems [142].

Solving this kind of problem is not trivial due to its complex nature, but they allow to easily present its huge search space and the effects of how to address it. Figure 3.1 shows the search space of Example 3.1, which correspond to the different truth values, encoded as 0 and 1, that can be assigned to the variables. Note that the instantiation and value assignment order of variables plays a fundamental role in the effectiveness to find a solution.

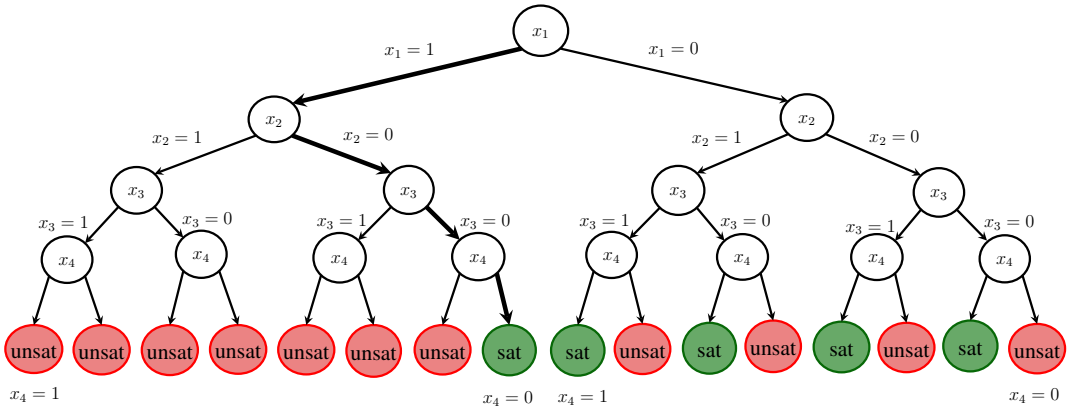



Figure 3.1 – Example 3.1 SAT formula search space.

Actually, most SAT solvers are based on Davis, Putman, Logeman and Loveland (DPLL) method [143, 144] from 1960s', which was retaken in 1990s' [3]. DPLL algorithm is a complete backtracking-based search procedure, which shrinks the search process by simplifying the CNF formula through the elimination of literals whose value could be easily derived in order to obtain a *true* value, e.g. literals in one-variable clause, or literals whose negation does not appears in the formula.

Despite being the smallest sensible case of first-order logic [3], the advances in SAT problem resolution have been boosted since 1990s [133] specially improving Automated Reasoning systems by using the Conflict-driven Clause Learning (CDCL) [145, 146, 147] algorithm, which adds a *back-jump* to the DPLL backtracking method. However, many problems are hard to map into a boolean logic space, because they need richer language to be modeled [4].

## 3.2 Satisfiability Modulo Theories: SMT

Satisfiability Modulo Theories, SMT [4, 148], is a generalisation of SAT born by the need of including and combining several first-order theories, called *modulos*, rather than boolean algebra theory, e.g. arithmetics or algebra theories. The inclusion and combination of these

modulos gives a richer language to reason and decide over more complex scenarios by modeling real-life situations more precisely [4].

**Example 3.2** Let be $a \in \mathbb{Z}$ an integer variable, defining the formula set $F = \{f_1, f_2\}$, with $f_1 = \neg(a \geq 3)$ and $f_2 = ((a \geq 3) \vee (a \geq 5))$. Is $\exists a \bigwedge_{i=1}^{2} f_i$ satisfiable? ♢

Note that in Example 3.2, SAT literals are replaced by formulas related to some theories, i.e., Linear Integer Arithmetic. Also, some additional first-order logic elements could appear, i.e., Universal and/or Existential variable quantifiers [1]. Then, through using a bidirectional mapping from boolean to theories space and using some theories-related techniques, called *Theories Solvers*, satisfiability procedure can be developed.

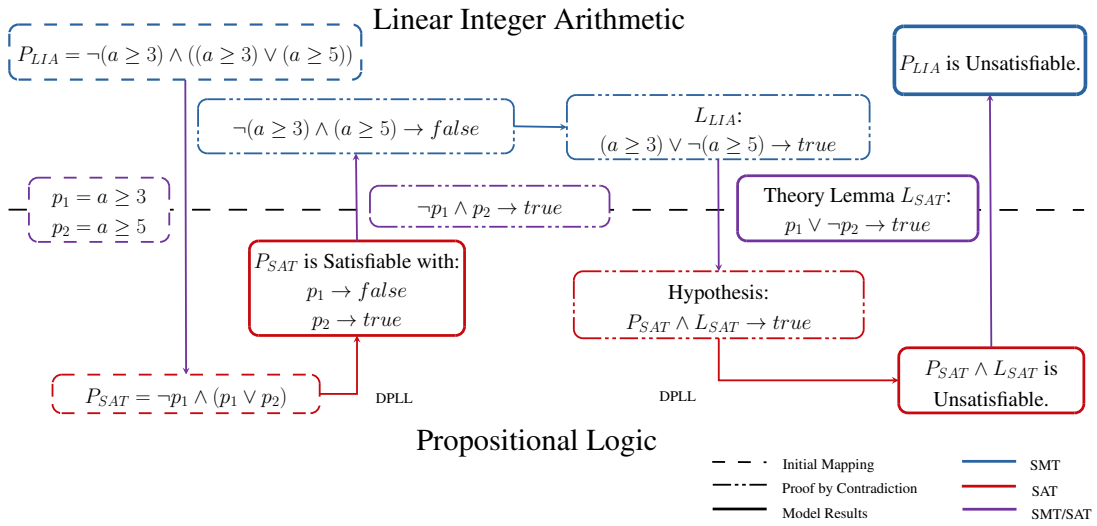



Figure 3.2 – Example 3.2 SMT formula solving procedure through DPLL(T) framework.

Despite generalizing SAT, SMT uses it as the fundamental core for solving its own formulas by integrating DPLL procedure with theories solvers, in a theoretical framework called **DPLL(T)**. Figure 3.2 summarises the resolution of the SMT formula shown in Example 3.2 by using DPPL(T) procedure:

1. SMT formula is mapped to SAT formula through the transformation of literals into boolean variables, i.e., $\neg(a \geq 3) \wedge ((a \geq 3) \vee (a \geq 5))$ into $\neg p_1 \wedge (p_1 \vee p_2)$.
2. SAT formula is solved using DPLL, finding a model for the mapped formula: $\{p_1 \rightarrow false, p_2 \rightarrow true\}$.
3. If SAT formula is unsatisfiable, then SMT formula is unsatisfiable. Otherwise, the solution model must be contrasted in the modulo scope by the theory solver.

1. In SAT problems all variables are assumed to be existentially quantified.

4. SAT model implies a set of literals to proof satisfiability, $\{\neg p_1, p_2\}$, but in the original modulo the literal set $\{\neg(a \geq 3), (a \geq 5)\}$ is unsatisfiable.
5. To check this inconsistency, a theory solver achieves a proof by contradiction using a *theory lemma*: as $\neg(a \geq 3) \wedge (a \geq 5)$ is unsatisfiable, therefore $(a \geq 3) \vee \neg(a \geq 5)$ is valid. Consequently, adding $p_1 \vee \neg p_2$ to the SAT mapped formula should not affect the obtained satisfiability model.
6. As the theory lemma inclusion leads to an unsatisfiable SAT formula, $\neg p_1 \wedge (p_1 \vee p_2) \wedge (p_1 \vee \neg p_2)$, then original SMT formula is also unsatisfiable in the integer arithmetic modulo.

As mentioned in [4], lemma proof procedure is repeated until convergence, and it always converges because a there a finite number of theories lemmas that could be created using the atoms in the formula.

### 3.2.1 SMT Applications

SMT great development in the last decade has been pushed by its adaptive capacity to model different situations. This has allowed applying SMT in different where uninterpreted first-order logic formulas would be too general or SAT formula would require additional encodings. These application include:

— Interactive Theorem Provers: Automated Theorem Proving (ATP) applications include the generation of formal proofs over mathematical hypothesis. One of these trends is Interactive Theorem Proving, which uses human-machine interaction to build the desired proofs. As well as SAT, SMT solvers have been applied to automatise some proofs in proofs assistants such as Dafny [149], HOL4 [150], and Isabelle [151].

— Constraint Satisfaction Problems: As SAT is the most famous CSP and have been used to model several Combinatorial Optimisation problems, but boolean modeling is a harsh constraint to adapt complex real life situations. SMT opens opportunities to solve these kinds of problems, especially the ones related to Constraint Programming, Constraint Satisfaction Problems (CSP). Examples include the use of SMT in Scheduling [133, 152, 153] and in Planning [154, 155, 156].

— Search-Based Software Engineering: First-order logic defines the ground to set different theories by symbols, operators and axioms. Several of these theories are basic elements that defines program building process in Software Engineering [4]. Therefore, several types of Search-based Software Engineering (SBSE) problems could be modelled thanks to formulas with different theories. SMT theories related to Software Engineering includes Linear and Real Arithmetics, Floating Points, Arrays, Bit-vectors, Boolean and Pseudo-Booleans theories and their combinations. These theories are applied in the fol-

lowing SBSE problems [3, 4, 9]: Program Verification, Symbolic Execution Testing, Program Analysis, Static Runtime Checking, Test Case Generation and Software Modeling.

Note that in several applications, theories could appear at same time and be related. SMT solving procedures handle the combination of multiple theories within a unique solving framework using the well-known Nelson-Oppen combination approach [157].

### 3.2.2 SMT-LIB

As the applications of SMT started to grow, an initiative rose in order to unify the interested community: SMT Library or **SMT-LIB** [158]. Since its inception in 2003, the initiative has pursued the following concrete goals:

— Develop and promote common input and output languages for SMT solvers, called **SMT-LIB standard** [159].

— Provide standard rigorous descriptions of background theories used in SMT systems, called **SMT Logics** [160].

— Establish and make available to the research community a **large library of benchmarks**.

— Collect and promote software tools useful to the SMT community. This is achieved through an annual competition called *The Satisfiability Modulo Theories Competition* or **SMT-COMP** [161].

#### 3.2.2.1 SMT-LIB standard

The SMT-LIB standard [159] (currently version 2.6) defines concepts, formal languages, and a command (script) language. It also introduces the concepts of *Theories* and *Logics* in order to classify problems or instances. A problem belongs to a logic; a logic refers to some theories; and a theory is a specific set of symbols together with a set of axioms that defines a well-known system.

#### 3.2.2.2 SMT Logics

As explained before, a SMT Logic (from now *logic*) is a classification for SMT problems or instances. A logic could refer to one or more modulos, therefore includes all possible combination of first-order theories. Figure 3.3 summarises some of the recurrent logics in SMT problems.

Logics have been named using letter groups that evoke the theories used and some restriction in their formulas. Some of the conventions are the following:

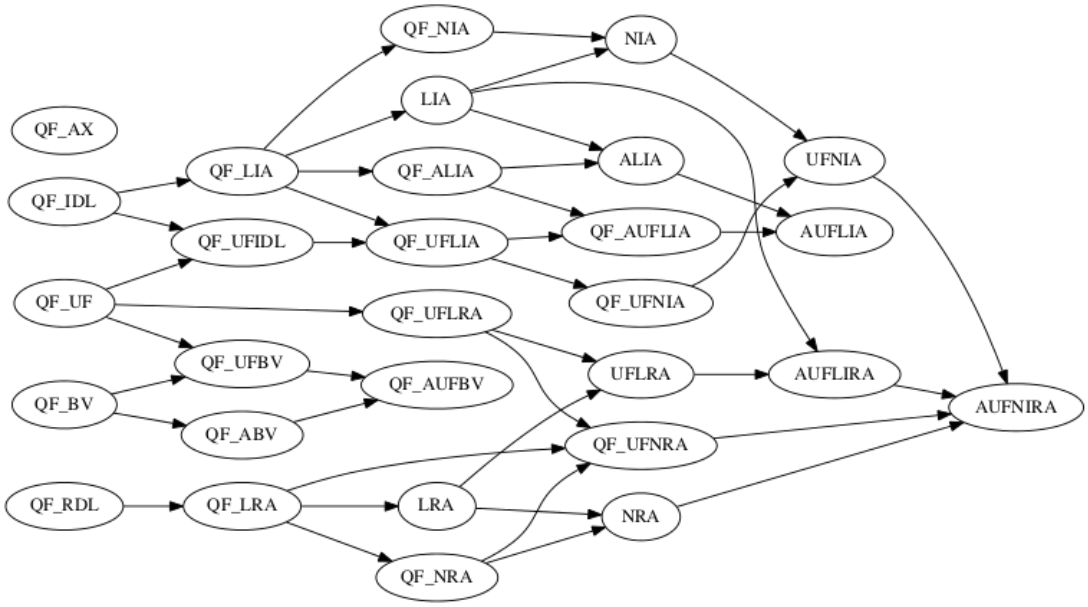



Figure 3.3 – SMT Logics [160]: SMT problem classification.

- `QF`: restricted to quantifier-free formulas, i.e., without universal or existential quantifiers.
- `A` or `AX`: Theory of Arrays.
- `BV`: Theory of Bit Vectors.
- `FP`: Theory of Floating Point.
- `IA`: Theory of Integer Arithmetic.
- `RA`: Theory of Real Arithmetic.
- `IRA`: Theory of Mixed Integer and Real Arithmetic.
- `L`: Linear fragment of an arithmetic logic.
- `N`: Non-linear fragment of an arithmetic logic.

Thus, logics related with more than one modulo are easily identified, e.g., `QF_ALIA` logics indicates SMT instances or problems with quantifier-free formulas mixing Linear Integer Arithmetic and Array theories. Note that most of Search-based Software Engineering problems are encoded as SMT problems related to the following logics: `LIA`, `LRA`, `QF_LIA` and `QF_LRA` [4]. We focus on this types of problems in this thesis.

#### 3.2.2.3 SMT-COMP

The Satisfiability Modulo Theories Competition or SMT-COMP [161] is an annual competition [162, 163, 164, 165, 166], born in 2005 [167], whose objective is to collect and promote software tools for solving SMT problems, but also to spur adoption of the common, community-designed SMT-LIB standard, and to spark further advances in SMT. SMT-COMP is inspired by others related competitions, as CASC [168] or the SAT competition [169], which have helped inspire continuing improvements in SMT tools.

#### 3.2.2.4 SMT Solvers

Automatic theorem provers for SMT, or **SMT solvers**, are software designed as tools to automatically decide the satisfiability of a given formula related to a set of theories. Generally, a solver core relies on the DPLL(T) procedure explained in Section 3.2. However, many SMT formulas are not suitable for being directly solved by it: they cannot be easily mapped into a SAT scope or the resulting SAT formula is too complex to be solved by DPLL.

**Example 3.3** Let be $x, y \in \mathbb{R}$ two real variables. A formula set $F = \{f_1, f_2\}$, where $f_1 = ((x \geq 1) \vee (y \leq 0))$ and $f_2 = (x + y = 0)$. Is $\forall x \exists y \bigwedge_{i=1}^{2} f_i$ satisfiable? ♢

Note that in Example 3.3, SMT formula is a `LRA` instance whose modulo defines operators symbols, as $+$ in $f_2$. Therefore, a SMT solver needs to combine several algorithms or heuristics before use its core resolution process in order to improve solver performance, e.g. reducing complexity of the formula through theory-related techniques. Hence, a SMT solver must define and decide a **strategy**: *how to select and apply these solving components?*.

Several solvers have been developed through SMT rise in Computer Science. Most of them have an ad-hoc design for a limited set of theories, e.g. MathSAT [170], OpenSMT [171]; while others have been discontinued but its contribution is an important milestone in the state of the art, e.g. BarceLogic [172]. Currently, three solvers are the most popular and used tools in SMT, sharing the features of cover most defined logics in SMT-LIB [7] and dominate the last five SMT-COMP.

— Z3 [6]: Microsoft Research's SMT solver. Its development was targeted at solving problems that arise in software verification and software analysis. It has been considered as the overall most reliable solver by winning SMT-COMP from its beginning in 2007 until 2017 (since 2014, it has been the symbolic winner because its non-competitive participation). Z3 takes advantage of an open and strong *strategy design* which helps to drastically improve performance of Z3. More detailed information is given in Section 3.3.

— CVC4 [7]: The Cooperating Validity Checker is a open-source joint project of the New York University and the University of Iowa. CVC4 has been the official winner of SMT-COMP from 2014 onwards. In the last competition it obtained, for first time in the last decade, a better overall performance than Z3 over all logics. Also, it have participated in SMT-COMP since the first edition.

— Yices [8]: The SRI International's SMT solver. It has been developed since 2006 and has obtained the second in the last five editions of SMT-COMP.

Note that all current SMT solvers uses SMT-LIB standard v2.X [159] as input/output language for modeling and solving SMT instances.

## 3.3 The Strategy Challenge in SMT

In 2013, De Moura and Passmore introduced *The Strategy Challenge in SMT* [10], which states the following:

> *To build theoretical and practical tools allowing users to exert strategic control over core heuristic aspects of high-performance SMT solvers.*

The main idea is to efficiently address the heuristics components in SMT solving, which are outside the scope of the DPLL(T) procedure, i.e., efficient creation of *strategies* to improve SMT solvers performance. To encourage the development these proofs and/or tools, Z3 provided a code-language interface to define **strategies**, i.e., a representation of the selection and ordering of modulo solving components. To proof the importance of the proposed challenge, Z3 strategy language was tested over `QF_LIA` logic using an incremental building. The final generated strategy reduced in 75% the amount of instances that could not be decided and around 60% the execution time.

This challenge has only been addressed by Graham-Legrand [173]. The proposal aims to use PSYCHE system [174], which allows users to test various techniques and strategies for either interactive or automated theorem proving. PSYCHE provides users with an API to use strategies as plugins for theorem proving solvers. Then, a Slot Machines approach [173] is introduced to ensure output correctness by using a recursive procedure, allowing test several strategies without jeopardise the processes of a selected theorem prover.

### 3.3.1 The Strategy Challenge as a SBSE problem

As explained in Chapter 1, the initial foundation of this work is inspired by the Search-Based Software Engineering (SBSE) challenge *Search for strategies rather instances* [2], which looks for strategies to solve classes of problems in SBSE rather than solving specific problem instances. Then, our hypothesis is to demonstrate that the search space of most SBSE problems could be efficiently handled using a hybridisation between a systematic search trend and a local search system. SMT is an appropriated systematic search system to use over SBSE problems given the natural concordance existing between first-order logic and software systems [3], and the current amount of SBSE problems, which are currently addressed with it [4]. Meanwhile, metaheuristics are the most used local search techniques in SBSE, especially Evolutionary Algorithms [129, 132].

But, ***how hybridise metaheuristics with SMT?*** The Strategy Challenge acts as integrative point between both trends. With regards to Z3, the creation of a modifiable interface adds degrees of freedom to create strategies outside the solver normal execution scope to improve its performance. This interface is based in a language, which is defined over a grammar, then the

creation or modification of these code segments could be done following some grammatical rules. This could be seen as Software Improvement problem (e.g. Program Refactoring, Transformation and/or Parameter Tuning), i.e., look for improved version of an existing software, where Genetic Improvement (see Section 2.2.1) is a well-known trend to solve these kind of SBSE problems. Therefore, metaheuristics techniques could be used to **automatically** search an improved SMT solver version related to a specific logic. Figure 3.4 shows the hybrid system, between a SMT solver and Metaheuristics, to address SBSE problems.

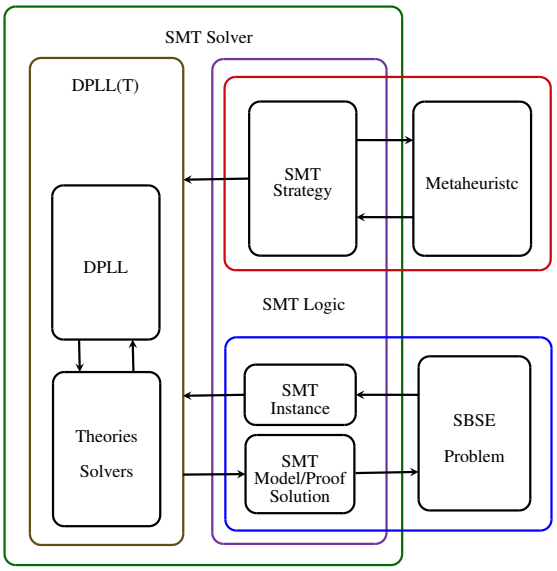



Figure 3.4 – A SMT-Metaheuristics hybrid system: Mixing SMT and Metaheuristic to solve SBSE problems.

Note that we reduced our initial task to a SBSE problem: *to found a good strategy configuration*, in order to improve overall SMT solver performance for an specific logic. This hybrid scheme have a great advantage: it is an abstraction level over our initial goal, i.e., **is not only interesting for SBSE problems encoded as SMT instances, but for all SMT applications**.

Regrettably, this hybrid approach has an important issue to address. Metaheuristics performance to generate efficient strategies will depend on the complexity of the problems and the SMT logic related to them. Thus, a basic metaheuristic, as Tabu Search, could work for some problems related to a specific logics, but fail for more complex ones. We have fall on the Algorithm Selection Problem [75]. We introduce an *Autonomous System* [76] approach for the generation of strategies, in order to ensure the selection of best available metaheuristic for address the instances of a selected SMT logic.

### 3.3.2 Autonomous Generation of Strategies

In addition to the discussion in Section 3.3.1, the idea of exert strategic control over SMT solver heuristics components adds a new limitation to the equation: *expert knowledge*. In spite

of the degrees of freedom added to the SMT solving process by creating strategies, the Strategy Challenge needs several expert guidelines to create effective tools to improve the decidability of a SMT solver, otherwise it will not achieve any advantage. Note that most of the time, users do not have the required knowledge in order to use properly all the heuristics features in SMT solvers, but some hints exists in default strategies that SMT provers use. These default strategies are generally designed by developers, or experts researchers, allowing us to use this information as guide for improvement. This lack of knowledge is a fundamental milestone of this research, aiming to take full advantage of the strategy structure in SMT solvers without need of expert knowledge.

In this thesis, we address the Strategy Challenge in SMT defining **a framework for the automatic generation of strategies for Z3 using an autonomous search system**, i.e., a practical system to automatically generate SMT strategies that improve SMT solvers performance *with no need of expert knowledge*.

Figure 3.5 resumes the architecture of this system applied to solve SBSE problems.

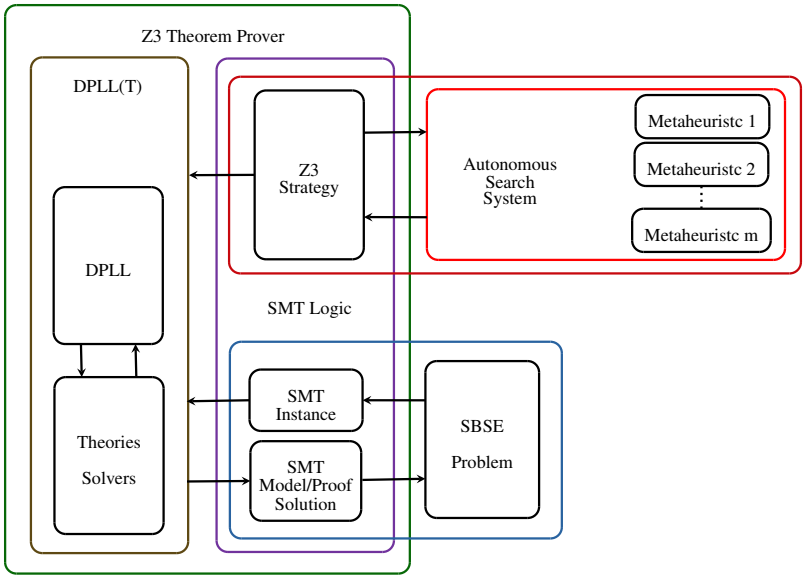



Figure 3.5 – Autonomous Generation of Strategies in SMT: Addressing the Strategy Challenge in SMT to improve SMT solving for SBSE problems with Z3.

## 3.4 Conclusions

In this chapter, Satisfiability Modulo Theories (SMT) concepts, instances and applications were introduced, as well as its fundamental basis, the Boolean Satisfiability Problem (SAT). We have presented the solving technique for SMT problems: DPLL(T), a combination between DPLL solving procedures used in SAT, and Theories Solvers, which are obtained from mapping the original SMT instance to SAT scope.

Community research efforts were also mentioned, being the SMT-LIB initiative (standard and benchmarks), the SMT logic classification, and the annual SMT Competition (SMT-COMP) the most relevant for our work.

Later, we sort out SMT solvers, software used to automatically to deduce satisfiability over SMT formulas, focusing in Microsoft Research's Z3 Theorem Prover, one of the most popular and efficient SMT solvers according to SMT-COMP.

Also, we analysed the Strategy Challenge in SMT, and how is the starting point of this research. We made emphasis in the concept of **strategies**, solving heuristics used outside the DPLL(T) scope to improve SMT solver performance.

Finally, we propose to address the Strategy Challenge in SMT through an Autonomous Search system in order to *automatically generate SMT strategies that improve Z3 solver performance without expert knowledge requirements*. This is not only interesting for SBSE problem mapped as SMT instances, as intended at first, but for all SMT related applications.

# 4

# Strategies

In the previous chapter, the *strategy* concept and the Strategy Challenge in SMT were discussed. However, the definition of strategy is still vague. This chapter address the importance of strategies in SMT, applied to the Z3 theorem prover. We deepen the definition of a strategy, for later analysing its structure in Z3 and how it helps to improve solver performance.

## 4.1 What is a Strategy?

The *strategy* concept is hard to define, because it changes depending on the application field and its boundaries are commonly blurry. A good start point is a neutral definition given by a well-known english language dictionary [175]: *a planned series of actions for achieving something*. With regards to Computer Science, the term strategy appears frequently. Here are some examples:

— Software Engineering: In Design Pattern [176], a strategy is a dynamic on-line algorithm selection in order to realise an operation depending on the requirements of the process.

— Evolutionary Computing: In Evolutionary Strategies algorithms [35, 36], a strategy is a scheme which define how a population evolves through generations.

— Hyper-heuristics: The automated selection or generation process of a heuristic from a pool of recommended procedures for a determined problem, is called strategy [177].

Sometimes, the concept is simply reduced to *a method of doing something or dealing with a problem*, being a synonym of the term *approach*. But, as in Computer Science is increasingly

common define problems as combinatorial search, a strategy could be defined as *adaptations of general search mechanisms which reduce the search space by tailoring its exploration to a particular class of problems* [10].

The last definition seems appropriated. However it does not consider the final goal of strategies, as in the SMT case context: optimise the performance of a major structure. Therefore, and considering all these perspectives, we define a strategy as: **a set of heuristics processes that helps to reduce the search space and/or the way how it is explored in order to find well-known solvable instances in a set of problems**.

### 4.1.1 Strategies in SMT

As explained in Section 3.3, we refer to a strategy in SMT as the way that several theories-related algorithms or heuristics are used; i.e., selected, arranged, and/or applied; outside the DPLL(T) solving process in order to improve SMT solver performance. These strategies appears implemented into the source-code of most SMT solvers and they are based in developers knowledge. The main goal of these heuristics methods or strategies is to reduce the complexity of the SMT formula, i.e., reduce formula search space, generating more efficient problem resolution.

With the appearance of the Strategy Challenge in SMT [10], the focus was fixed into the importance of the strategies for the behaviour of SMT solvers, especially in the Z3 theorem prover, which, at date, were relevant only for developers. Then, according to the needs of the SMT solver users, build or modify a strategy shows empirically the impact of the selection, arrangement and application of heuristics elements to address as best as possible a set of problems.

## 4.2 Z3 Strategies

In Z3 theorem prover, strategies could be found inside its source-code as well as they could be created through a language designed to represent the way heuristics and formula solving techniques are applied. Strategies define the way heuristics or solving components are applied over a SMT formula in Z3, regardless where they were defined, by following *the tactic framework*.

### 4.2.1 The Tactic Framework

The tactic framework define how strategy components, called *tactics*, interact with a SMT formula in order to reduce and/or solve it. Figure 4.1 shows the rules that define the tactic framework.

$$
\begin{array}{lcl}
goal & = & formula\ sequence \times attribute\ sequence \\
tactic & = & goal \rightarrow return \\
return & = & empty \rightarrow model \\
 & | & false \rightarrow proof \\
 & | & goal \rightarrow goal\ sequence \times modelconv \times proofconv \\
 & | & fail \\
modelconv & = & model\ sequence \rightarrow model \\
proofconv & = & proof\ sequence \rightarrow proof
\end{array}
$$

Figure 4.1 – The Z3 Tactic Framework: Solving formulas using tactics.

As explained above, a strategy is composed of elements called *tactics*. A *tactic* could be a heuristic or a solving component, as well as a combination of some of them. Even if two tactics can be different, or focused on separated task, they are treated in as the same: *a set of rules or constraint* to be applied. These tactics operate over a SMT formula, called *goal*. A *goal* is a sequence of modulo-related formulas which includes a set of specific attributes. When a tactic is applied over a goal, a subproblem set is always returned. The possible subproblem set values are:

1. *Empty Set*: when a tactic determines if a goal is satisfiable, a empty set is returned. Tactics applied to this set do not modify it. The empty set is related to a *model* that proves the satisfiability of the goal.
2. *False set*: if a tactic computes an unsatisfiable goal, a false set is returned. As in the empty set, any tactic applied to the false set does not change the output. It has associated a unsatisfiability proof for the goal.
3. *Goal set*: when a tactic is applied to a problem, but neither empty nor false set is reached, the returned subproblem set can be:
   (a) the original problem $G$, if the applied tactic does not change the initial goal;
   (b) a subset $G'$, which represent the original goal $G$ after apply a tactic procedure. It has associated a model converter (*modelconv*) and a proof converter (*modelconv*), which can rebuild a satisfiability model or unsatisfiability proof, respectively, for the original goal $G$ form the subset $G'$, if a solving result is found.
4. *Fail set*: if the tactic does not have all requirements to properly work, a fail set is returned and the original goal is not processed. This may have two consequences:
   (a) it leads to a global fail result if tactics are combined by conjunction, which means that no result could be obtained.
   (b) it skips the failing tactic if tactics are joined by disjunction, using the next tactic over the original problem or goal $G$.

Note that Z3 as well as most SMT solvers, must show the following results, defined in SMT-LIB standard [159], when a instance is addressed:

1. `sat`: when the problem is satisfiable. This value is related to the empty set.
2. `unsat`: when the problem is unsatisfiable. This value is related to the false set.
3. `unknown`: when no solution could be decided. This value is related to the fail set.
4. `timeout`: when solving time limit is exceeded before the final decision could be determined.

### 4.2.2 Tactics classification

As shown before, a tactic encapsulates different formula solving procedures under the same concept. To clarify their difference, we classify tactics in *basic tactics* and *compounded tactics* and we formalise them by using the notation shown in Table 4.1.

| Symbol | Definition |
|---|---|
| $\Phi$ | Set of all SMT goals. |
| $\Pi$ | Set of all parameter vector of tactics. |
| $\Lambda$ | Set of all satisfiability models and possible unsatisfiability proofs of a goal. |
| $I$ | Set of SMT formula solving outputs. $\{\texttt{sat}, \texttt{unsat}, \texttt{unknown}, \texttt{timeout}\}$ |
| $J$ | Set of boolean values. $\{\texttt{True}, \texttt{False}\}$ |
| $T$ | Set of all possible tactics. |

Table 4.1 – Notation for tactic formalisation

#### 4.2.2.1 Basic tactics

Basic tactics are atomic elements of a strategy. They can be used individually or in a compounded tactic. There are three types of basic tactics:

**Definition 4.1** A ***solver*** tactic checks the satisfiability of a goal. Any solver can be defined as:

$$S : \Phi \times \Pi \rightarrow I$$

Note that, the application of a solver $S$ relies in its own parameter vector $\alpha \in \Pi$ to generate goal $G \in \Phi$ output. $\diamondsuit$

**Definition 4.2** A ***heuristic*** tactic transforms the problem into a sequence of subproblems. Any heuristic can be defined as:

$$H : \Phi \times \Pi \rightarrow \Phi^n \times \Omega$$

where $\Omega$ is a satisfiability model or an unsatisfiability proof converter, from the generated subgoals to the original goal, defined as:

$$\Omega : \Lambda^n \rightarrow \Lambda$$

We define the application of a heuristic $H$ with parameter vector $\alpha \in \Pi$ to a problem or goal $G$ as:

$$H(G, \alpha) = ((G_1, G_2, \ldots, G_n), \Omega(G_1, G_2, \ldots, G_n)) \quad \diamondsuit$$

**Definition 4.3** A ***probe*** tactic checks if in its current state the problem or goal has some property. A probe is formalised as:

$$P : \Phi \rightarrow J$$

Note that a probe $P$ applied to a goal $G$ returns a truth value depending on the existence of some goal property. $\diamondsuit$

#### 4.2.2.2 Compounded tactics

A ***compounded*** tactic is a combination of tactics through the use of ***tacticals***.

**Definition 4.4** A ***Tactical*** is a function that define how tactics are applied and/or combined. Using a tactical over a set of tactics generates *a new complex tactic*. Tacticals are defined as:

$$C : 2^T \rightarrow T$$

Thus, a tactical $C$ over a set of tactics $\tau \in T$ generates a new tactical $t \in T$. $\diamondsuit$

## 4.3 Formalising Z3 Strategies

Once explained all strategies components, introduced by the tactic framework, we could formalise the notion of Z3 strategy through a language which compiles into the tactic framework and define a strategy grammar. For introducing the grammar, we use the following notations based in first-order logic systems:

— Arity function: $ar(f)$ is the arity of the function symbol $f$, e.g., if a function symbol $f$ is defined as $f(,)$, then $f$ has arity $ar(f) = 2$.

— Function symbol: Let $f$ be a function symbol, then $f_{/n}$ refers to $f$ with arity $ar(f) = n$.

— Constant: Let $c$ bet a function symbol, then $c$ is a constant iff $c$ has arity $ar(c) = 0$.

### 4.3.1 Strategy syntax components

We consider the following sets in order to build strategies: Constants, Functions, and Parameters.

#### 4.3.1.1 Constants

Let $\Theta$ be a set of constants $\Theta = \texttt{Solver} \cup \texttt{Heuristic} \cup \texttt{Probe}$ where `Solver`, `Heuristic`, and `Probe` are sets of constant functions that correspond to the various types of basic tactics explained in Section 4.2.2.

— The set `Solver` contains procedures to check the satisfiability of a problem (or subproblem).

— The set `Heuristic` contains techniques that transform a problem into a sequence of subproblems.

— The set `Probe` contains functions to check properties of the current subgoal of a problem.

#### 4.3.1.2 Functions

Strategies in Z3 includes two types of functions: combinators and parameters modifiers.

##### 4.3.1.2.1 Parameters modifiers

Let $\Delta$ be the set of binary functions which allow to change the parameter vector values of a tactic. A function $\delta \in \Delta$ is defined as:

$$\delta : T \times \Pi \rightarrow T$$

Note that given a tactic $t \in T$ and parameter configuration $\alpha \in \Pi$, a $\delta$ function creates a new tactic $t' \in T$. Consequently, this set is composed by the following functions:

— $\texttt{try-for}_{/2}$: function that defines the running time-limit of a tactic in milliseconds.

— $\texttt{using-params}_{/2}$: function that defines parameters values to a tactic, e.g., random seed.

##### 4.3.1.2.2 Combinators

Let $\Gamma$ be the set of n-ary functions for combining tactics, i.e., generate new complex tactics. A combinator $\gamma \in \Gamma$ is defined as:

$$\gamma : T^n \rightarrow T$$

Note that given a set of $n$ tactics $T' = \{t_1, t_2, \ldots, t_n\}$ with $T' \subset T$, a combinator $\gamma$ generates a new composetd tactic $t' \in T$. This set is the union of the following functions set:

— $\Gamma^{and} = \{\texttt{and-then}_{/n} | n \geq 1\}$, where each function combines conjunctively a set of $n$ tactics.

— $\Gamma^{or} = \{\texttt{or-else}_{/n} | n \geq 1\}$, where each function combines disjunctively a set of $n$ tactics.

#### 4.3.1.3 Parameters

Let $\Pi$ be the set of parameter vectors, considered as constants, which correspond to tactics parameters. As shown in Section 4.2, only elements from `Solver` and `Heuristic` set have a parameter vector $\alpha \in \Pi$ by their definition.

However, a combinator function $\gamma \in \Gamma$ generates a *composed tactic* formed from a set of tactics $T' = \{t_1, t_2, \ldots, t_n\}$ that includes constants from $\{\texttt{Solver} \cup \texttt{Heuristic}\}$. Thus, a $\gamma$ function could have associated a parameter vector $\beta$ to be applied to each tactic $t_i \in T'$.

Therefore, we can apply a parameter function $\delta \in \Delta$ to a $\gamma$ function, making $\delta(\gamma, \beta)$ is valid. This beacause, $\beta$ will be applied to all tactics in the combinator with $\delta(\gamma, \beta) = \gamma(\delta(t_1, \beta), \delta(t_2, \beta), \ldots, \delta(t_n, \beta))$. Note that, this expression is also valid.

### 4.3.2 Strategy language grammar

As usually, we represent strategies as trees, i.e., first-order terms. The set of terms $T(\Sigma)$ is built on the signature $\Sigma = \Theta \cup \Delta \cup \Gamma \cup \Pi$. Of course not all strategies are correct with regards to the syntax and semantics of Z3. Based on the Z3 strategy syntax components explained in Section 4.3.1 and in order to restrict the set of allowed strategies, we use a term grammar [178].

A term grammar is a tuple $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{S}, \mathcal{P})$, where $\mathcal{N}$ is the set of non-terminal symbols, $\mathcal{T}$ is the set of terminal symbols, $\mathcal{S}$ is the starting symbol, and $\mathcal{P}$ is the set of production rules.

**Definition 4.5** The following term grammar, $\mathcal{G}_{Z3}$, represents Z3 strategies:

— $\mathcal{N} = \{\texttt{Tactic}, \texttt{CStrategy}, \texttt{Strategy}\}$

— $\mathcal{T} = \Sigma$

— $\mathcal{S} = \{\texttt{Strategy}\}$

— $\mathcal{P}$ includes the following rules:

(a)
$$\begin{array}{lcl} \texttt{Strategy} & \rightarrow & \texttt{CStrategy} \\ & | & \texttt{Tactic} \end{array}$$

(b)
$$\begin{array}{lcl} \texttt{CStrategy} & \rightarrow & \delta(\texttt{Strategy}, \pi); \text{ with } \delta \in \Delta, \pi \in \Pi \\ & | & \gamma(\texttt{Strategy}, \ldots, \texttt{Strategy}); \text{ with } \gamma \in \Gamma, ar(\gamma) = n \end{array}$$

$$
\begin{array}{lll}
\texttt{Tactic} & \rightarrow & p\text{; with } p \in \texttt{Probe} \\
\text{(c)} & | & h\text{; with } h \in \texttt{Heuristic} \\
 & | & s\text{; with } s \in \texttt{Solver}
\end{array}
\qquad \diamondsuit
$$

The language generated by this grammar is denoted `Strat` and corresponds to well-formed strategies, i.e., syntactically correct terms. For better understanding, we refer the reader to the following example.

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

Figure 4.2 – User-defined strategy example.

**Example 4.1** Let be $S$ an user defined strategy, shown in Figure 4.2. The strategy $S$ is a composed tactic, generated by using the conjunctive combinator `and-then` over the following tactics:

— A probe, `fail-if(not(is-ilp))`, which checks if the SMT formula is a integer linear programming model.

— A heuristic, `simplify`, which reduces the SMT formula complexity by applying context-independent simplification rules, e.g. constant folding, $(x + 0) \rightarrow x$.

— A heuristic, `split-clause`, which separates clauses of literals related by disjunction ($\vee$) into cases. e.g. $G : (x > 0) \wedge ((y < 1) \vee (y > 1))$ into $G_1 : (x > 0) \wedge (y < 1)$ and $G_2 : (x > 0) \wedge (y > 1)$.

— A composed tactic of solvers, joined disjunctively through the combinator `or-else`. This tactic includes:

  — The solver `sat`, which try to decide over a SMT formula using a SAT solver. This solver has defined, through the use of the binary function `try-for`, a time limit of `100` milliseconds.

  — The solver `smt`, which try to decide over a SMT formula using a solver based in DPLL(T) procedure. This solver has defined, through the use of the binary function `using-params`, a specific value for the solver `random seed` parameter of `100`.

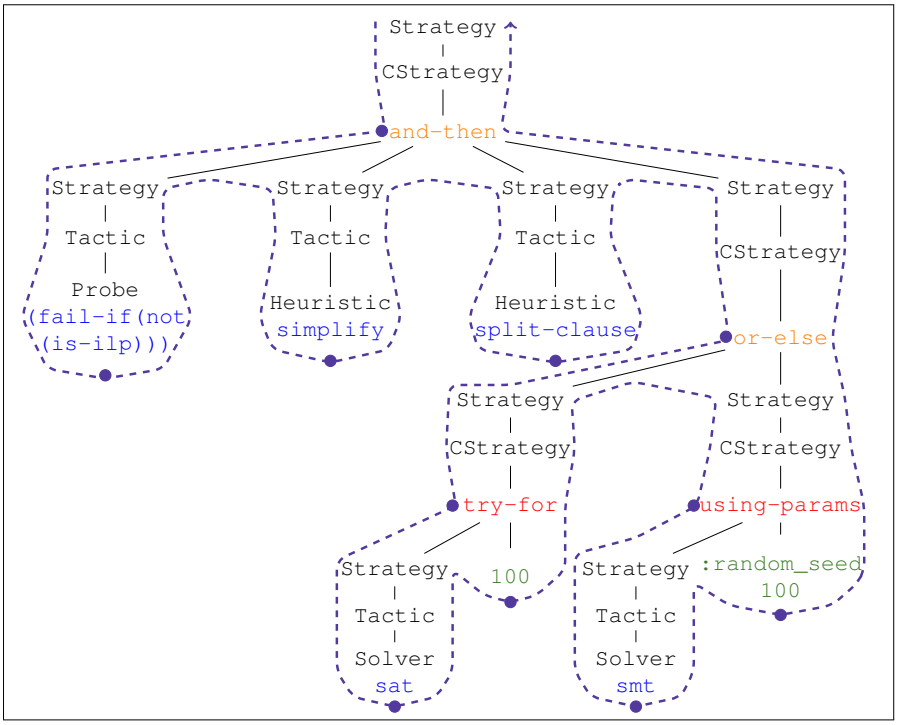



Figure 4.3 – User-defined strategy generated by the derivation tree of the Z3 strategy term grammar.

The strategy $S$ can be represented by the term $st \in \texttt{Strat}$ generated by the grammar $\mathcal{G}_{Z3}$, and explained by the derivation tree shown in the Figure 4.3. Note the color used: blue code represent a basic tactical, orange sentences indicates n-ary combinators functions, red refers to binary functions, and green expressions show values modified in the parameter vector of a tactic. ♢

## 4.4 Using Z3 Strategies

In order to show how to use strategies and how they perform to decide over a SMT formula, let us introduce the following example to be used for a step by step explanation.

**Example 4.2** Let be $a \in \mathbb{Z}$ an integer variable, which define the clause set $F = \{f_1, f_2\}$ with $f_1 = ((a < 10) \vee (a = 10) \vee (a > 10))$ and $f_2 = (2a = 20)$. Is $\exists a \bigwedge_{i=1}^{2} f_i$ satisfiable?♢

### 4.4.1 SMT-LIB standard language

The SMT-LIB standard [159] defines an input language for formulas to be understood by SMT solvers. Its syntax is based in first-order sorted languages, where operators symbols appear

at first in an expression, to then mention the elements that will be related, e.g. the clause $(x + y) \geq 1$ will be expressed as `(>= (+ x y) 1)`.

```
1 (declare-const a Int)
2 (assert (or (< a 10 ) (= a 10) (> a 10))
3 (assert (= (* 2 a) 20))
4 (check-sat)
```

Figure 4.4 – SMT-LIB standard example: A simple problem in Integer Arithmetic Modulo Theory.

Figure 4.4 shows the integer arithmetic modulo formula in SMT-LIB standard language. At first, variables (as well as functions) must be defined, e.g. in line 1 a constant symbol `a` (i.e., variable) in the theory of integer arithmetic is defined. Thus, the problem will belong to a SMT logic that includes all modulos defined in variables and functions. After symbol definition, each clause is expressed using the `assert` token, following the standard syntax rules. Line 2 refers to the clause $f_1$, as line 3 refers to clause $f_2$. The command `check-sat`, indicates that all clauses were defined and the solver could decide over the formula, i.e., the conjunction of all clauses, using the default strategy defined in the solver for the logic of the problem.

### 4.4.2 Adding strategies

Syntactically, Z3 includes two ways to apply a strategy in a formula modeled with SMT-LIB standard, as shown in Figure 4.5. In the first case, Figure 4.5a, an end-user strategy is defined before the `check-sat` instruction. Thus, the user-defined strategy will be applied at first, to then, if necessary, perform the Z3 default strategy. In the second case, Figure 4.5b, the Z3 default strategy is completely replaced by the end-user strategy. Note that `check-sat` instruction is replaced by `check-sat-using` token. This new instruction, also set the end of clause definition, but also implies that SMT formula will be decided using an user-defined strategy. Since we want to create new alternatives to Z3 default strategies, in this work we use the second syntax in the procedures of our system for autonomous generation of strategy.

```
1 <Problem header and assertions>
2 (apply <strategy>)
3 (check-sat)
```

(a) Conjunction between an user-defined *strategy* and default strategy.

```
1 <Problem header and assertions>
2
3 (check-sat-using <strategy>)
```

(b) Replacing default strategy with an user-defined *strategy*.

Figure 4.5 – Syntax to apply Z3 strategies over a SMT formula.

### 4.4.3 Solving formulas using strategies

Strategies define how to address the solving procedure. Thus, the way its components are selected and placed plays a major role in the Z3 performance.

```
1  (declare-const a Int)
2  (assert (or (< a 10 ) (= a 10) (> a 10))
3  (assert (= (* 2 a) 20))
4  (check-sat-using
5          (and-then
6                  (fail-if (not(is-ilp)))
7                  simplify
8                  split-clause
9                  (or-else
10                         (try-for sat 100)
11                         (using-params smt :random-seed 100)
12                 )
13         )
14 )
```

Figure 4.6 – Simple problem in Integer Arithmetic Modulo Theory featuring an end-user strategy.

**Example 4.3** Let be $G = \exists a : \{((a < 10) \vee (a = 10) \vee (a > 10)) \wedge (2a = 20)\}$ the goal defined by the SMT formula of Example 4.2. Also, let $S$ the strategy defined in Example 4.1, which represent the term $st \in \texttt{Strat}$ generated by the term grammar $\mathcal{G}_{Z3}$. Figure 4.6 shows the use of the user-defined strategy $S$ (lines 4 to 14) over the linear arithmetic formula $G$ (line 1 to line 3) in a SMT-LIB standard input for Z3 solver. ♢

In this problem, one probe, two heuristics tactics, and a disjunctive composed tactic, which includes two solvers tactics, are applied in a linear conjunctive order strategy (between lines 5 and 13). The solver interprets the strategy as follows:

1. At line 4, the `check-sat-using` command indicates the end of the clauses definition, and the begin of formula solving using an user-defined strategy.
2. The function `and-then` marks the begin of a composed tactic as strategy (line 5). This function joins conjunctively a set of tactics (line 6 to 9).
3. At line 6, the probe `is-ilp` is the first basic tactic to be applied. It checks if the problem is in an Integer Linear Programming (ILP) form. The result of the probe will be processed by the function `fail-if` as fail if the probe result is true, or as the original goal if it is false. Therefore, using the negation of the probe, `(not is-ilp)`, allows to apply the designed strategy only when the goal is in ILP form, which is this case.

4. The first applied heuristic (`simplify`, line 7), reduces the problem and gives the following subgoal:

$$G' = \exists a : \{(\neg(a \geq 10) \vee (a = 10) \vee \neg(a \leq 10)) \wedge (a = 10)\}$$

5. The heuristic `split-clause` (line 8) splits disjunctions into a subgoal set, and returns the following:

$$G'_1 = \exists a : \{\neg(a >= 10) \wedge (a = 10)\}$$

$$G'_2 = \exists a : \{(a = 10) \wedge (a = 10)\}$$

$$G'_3 = \exists a : \{\neg(a <= 10) \wedge (a = 10)\}$$

6. The combinator function `or-else` (line 9) generates a composed tactic, by joining disjunctively two solver tactics (line 10 and line 11).

7. First attempt of solving is done by `sat` tactic (line 10). It cannot solve or modify any subproblem within the timeout of 100 milliseconds specified in the tactical `try-for`. Hence, it returns the fail subgoal set. However, this tactic is inside a disjunctive combinator, thus the original subgoal set must be analysed by the following tactic.

8. The last tactic `smt` (line 11) uses DPLL(T) procedure in each goal, solving the whole set. It returns the following subproblem set: $G'_1 = false$, $G'_2 = empty$, $G'_3 = false$. The use of the tactical `using-params` allows to change the default value of the `random seed` generating a new parameter vector for the tactic.

The tree in Fig. 4.7 sketches the application of the strategy tactics. At the end, Z3 can rebuild the original problem in conjunctive normal form and returns an *empty set* as the final result. Finally, the translation of this subset is the expected `sat` output.

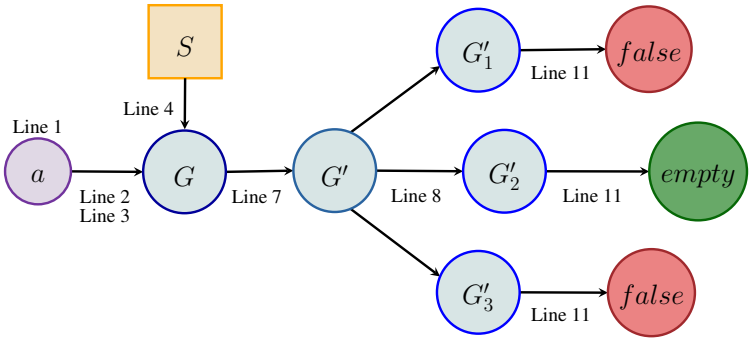



Figure 4.7 – Solving a SMT formula using an user-defined strategy: Linear arithmetic modulo arithmetic example. Information under arrows refer to line numbers of the example in Figure 4.6.

## 4.5 Conclusions

In this chapter, we defined the concept of *strategy*, given a search optimisation analogy. Also, we compared this concept with other related definitions. Thus, we understood why the selection, arrangement and application of heuristics components in SMT solvers is also called *strategy*.

Concerning to Z3, the solver used to present The Strategy Challenge in SMT [10], we analyzed the structure of its strategies. We introduce the tactic framework, which define a *tactic* as strategy core element, and how it interacts with a SMT formula. Then, we explained the different tactic types found in most strategies. This framework acts as a low-level language, a kind of assembly language used by Z3.

Moreover, we describe the Z3 strategy language, a high-level language that can be compiled into the tactics defined in the Z3 tactic framework. Then, we formalise their syntax components to define a term grammar. This grammar acts as constructor for every well-formed Z3 strategy.

Finally, we explained how strategies, in Z3, work over a SMT formula in a step-by-step resolution example.

# II

# Contributions

# 5

# Automated Generation of Strategies

Next, we address the task of strategy building. We analyse different search approaches to define an automated process for generating strategies. This procedure is used as core element to define our framework proposal for autonomous generation of strategies in SMT.

## 5.1 Search Paradigm Selection

As explained in Section 3.3.1, the strategy building process could be seen as a Search-Based Software Engineering (SBSE) problem, specifically a Software Improvement problem. To address this task, we should define what search paradigm to use. We focus on the paradigms exposed in Section 2.1.2.

### 5.1.1 Local Search over Systematic Search

Given the nature of the search space, the processes for automatic strategy generation are based on **local search procedures** leaving aside systematic search ideas. Systematic search procedure will tend to fail, based on the following remarks:

— Huge search space: Given the amount of valid strategies, which is a finite but intractable number, systematic procedures will not found or ensure the existence of an optimum strategy.

— No global optimal bound: As optimal values depends by the amount of problems or instance decided to solve, an this will change depending of the class to address, there is not a optimal bound to approach it.

— Progressive benchmarks: SMT instances grow up in function of time, therefore an (local) optimum strategy will lose it quality while new type of problems appears.

Hence, the use of local search procedures could be naturally adapted to this kind of problem. Also, evidence of its efficiency has been shown, especially using evolutionary approaches [40, 44, 45, 46, 47].

### 5.1.2 Perturbative Search over Constructive Search

Our system for automated generation of strategies follows **perturbative search guidelines rather than constructive search** by using evolutionary computing methods. Let us remark that generating strategies from scratch is possible, but rather intractable due to the size of the search space. This problem complexity is drastically reduced if the generation process repairs a well-known strategy.

Remember that our task is constrained by the idea of build a system for generate strategies without need of expert knowledge. This does not implies avoid the use of specialised strategies contained within SMT solvers. These strategies could be very useful to cut problem search space. Expert-defined strategies adds valuable information to the generation process that end-users does not handle, as:

— Strategies structures: It avoids the process of creating a strategy structure from scratch.

— Sub-strategies: Composed tacticals (or continuous portions of them) could be treated and analysed as a single tacticals or a single strategy. These could weight its contribution to the strategy purpose or its impact for a selected logic.

— Tactic arrangement: Once a structure is defined, perturbations could be more easily performed according to the tactic arrangement in the strategy. Thus, if we want add or replace an heuristic tactic, we already known where to locate it, and analyse it impact.

Thus, it is more effective to get default strategies to initialise our process in order to aim for better quality designed strategies.

## 5.2 The Rules Approach

We introduce a system based on the application of a set of rules as perturbative search procedure in order to generate new strategies. This *rules approach* constitutes the key feature of our strategy generation process. Rules are applied on strategies in order to modify either their structures or their behaviours. According to the Z3 strategy grammar defined in Section 4.3.2, we classify these possible modifications according to the elements that affect:

1. Structural components: the sets $\Theta$ (basic tactics) and $\Gamma$ (composed tactics) that define the structure of tactics and how tactics will be combined and applied.

2. Behavioral parameters: the set $\Pi$ of parameters that helps defining the specific behavior of the tactics, generally applied by functions of $\Delta$ set.

Note that changing any element from these sets can dramatically affect the solver performance. We will thus have some structural rules to modify structural components and behavioral rules to change values of behavioral parameters.

### 5.2.1 The Rules

Before formalizing how rules will be applied on strategies, we need to introduce classic term notations. Given a term $t \in T(\Sigma)$, $Pos(t)$ is the set of the positions in $t$ (labels of the nodes). Positions are classically labeled by words on $\mathbb{N}$ (i.e., sequences of integers) as the following:

— $\epsilon$: the position of the root.

— $p.i$: the position of the $i^{th}$ argument of the function symbol at position $p$.

— $t|_p$: the sub-term of $t$ at position $p$.

— $t(p)$ is the function symbol at position $p$.

— $t[t']_p$ is the term obtained by replacing the sub-term $t|_p$ by $t'$.

This notation will be used to define rules as classic term rewriting rules [179]. We identify two types of rules depending on the context of strategies: simple rules and specific rules.

#### 5.2.1.1 Simple Rules

We present *simple rules* as perturbations applied to strategies regardless of the context of themselves and the resulting strategies.

**Definition 5.1** A ***simple rule***, $r$, is defined as:

$$r : l \to m$$

where $l, m \in T(\Sigma)$, i.e., $l, m$ are terms built from the initial signature. To apply a rule $r$ over a strategy $st \in \texttt{Strat}$, the following condition must be satisfied:

$$st \xrightarrow{r} st[m]_p \iff p \in Pos(st) \mid st|_p = l$$

that is, the term $l$ exist in the strategy $st$ in the position $p$. In this case, the strategy $st$ is rewritten into the strategy $st'$ defined by:

$$st' = st[m]_p$$

Note that, $st' \in \texttt{Strat}$ is required in order to insure that rules are correct and generate only valid strategies. ♢

Moreover, some additional constraints may be required in order to apply simple rules.

**Definition 5.2** A ***constrained simple rule***, $r_c$, is defined as:

$$r_c : l \rightarrow m \ \{C\}$$

Note that $r_c$ rules inherit $r$ rules requirements, but also must fulfil the following condition:

$$st \xrightarrow{r_c} st' \ \texttt{iff} \ C \text{ is satisfied}$$

that is, $st, st', l$, and $m$ must satisfy the constraint $C$. Note that the constraint may be used to check some syntactical properties of $st$ or to insure required semantical properties. ♢

#### 5.2.1.2 Specific Rules

The application of some rules may sometimes be dependent of the context, i.e., the strategy $st$ on which the rule is applied or the resulting strategy $st'$. We call them *specific rules*.

**Definition 5.3** A ***specific rule***, $r_s$, is defined as:

$$r_s : st[l] \rightarrow st[m] \ \{C_s\}$$

where $C_s$ is a constraint that may involve the whole context, i.e., the entire resulting strategy, as well as, sub-strategies contexts. Also, $r_s$ rules inherits $r$ rules properties, and to apply a rule $r_s$ over a strategy $st$, the following condition must be satisfied:

$$st \xrightarrow{r_c} st' \ \texttt{iff} \ C_s \text{ is satisfied}$$

Note that, we omit explicitly mention the position in the rules for simplicity, because the origin ($st[l] \implies st|_p = l$) and the result ($st[m] = st[m]_p$) refer to the same place in the strategy. Of course, when the context is not necessarily required, a specific rule is equivalent to a constrained simple rule $r_c : l \rightarrow m \ \{C\}$. ♢

Moreover, some rules involve several strategies as inputs. Therefore we extend the previous rule notation for multiples strategies.

**Definition 5.4** A ***multiple specific rule***, $r_{ms}$, is defined as:

$$r_{ms} : (st_1[l_1], \ldots, st_n[l_n]) \rightarrow (st_1[m_1], \ldots, st_n[m_n]) \ \{C_{ms}\}$$

which is a tuple composed by $n$ rules of type $r_s$, restricted by a set $C_{ms}$ of $C_s$ constraints that must be satisfied. Therefore exists a tuple at positions $(p_1, \ldots, p_n) \in Pos(st_1) \times \ldots \times Pos(st_n)$ such that $\forall\ 1 \leq i \leq n,\ st_i|_{p_i} = l_i$ and $st[m_i] = st[m_i]_{p_i}$. ♢

#### 5.2.1.3 Notations used for defining the rules

With regards to the Z3 strategy grammar and its basic components, the rules applied to change strategies are defined using the following notation:

— $st[]$: a highlight of the context to the application of some rules, only if required.

— $st^n$: a n-ary sequence of strategies, i.e., $st^n \in \texttt{Strat}^n$, $st^n = st_1, \ldots, st_n$

— $\gamma_{/n}$: a combinator function of arity $n$, i.e., $\gamma_{/n} \in \Gamma$

— $\delta$: a parameter modifier function with arity 2, i.e., $\delta \in \Delta$

— $s$: a solver, i.e., $s \in \texttt{Solver}$

— $h$: a heuristic, i.e., $h \in \texttt{Heuristic}$

— $p$: a probe, i.e., $p \in \texttt{Probe}$

— $\pi$: a parameter vector, i.e., $\pi \in \Pi$

Any rule defined using these notations is thus a generic rule that can be instantiated to match a given strategy $st \in \texttt{Strat}$ by means of $\mathcal{G}_{Z3}$.

**Example 5.1** Let be C the rule defined as:

$$C : \texttt{and-then}_{/n}(st^n) \rightarrow \texttt{or-else}_{/n}(st^n)$$

This rule must be understood as a generic rule for any value of $n$ and any sequence of strategies $st^n$.

```
(and-then
        simplify
        (try-for sat 100)
)
```

(a) Strategy $T$

```
(or-else
        simplify
        (try-for sat 100)
)
```

(b) Strategy $T'$

Figure 5.1 – Modifying Strategies using Rules: Applying Example 5.1 rule $C$ in strategy $T$ generating strategy $T'$

As shows Figure 5.1, given the initial strategy $T$ (Figure 5.1a) is possible to obtain the resulting strategy $T'$ (Figure 5.1b) by using the rule $C$:

$$T \xrightarrow{C} T'$$

where $n = 2$ and $st^2 = h, \delta(s, \pi)$. Note that in our context, for sake of simplicity, we do not introduce variables nor substitution mechanisms as in classic rewriting systems. We rather work on closed terms (i.e., strategies) and consider that our rules are almost meta generic rules that can be instantiated to match the strategies to be improved. Rules may respect some conditions before being applied, which are modeled as constraints. ♢

## 5.2.2 Constraints

Some important requirements are mandatory when rules are used to transform strategies. Therefore, some *constraints* will ensure that these semantics requirements are satisfied when a rule is applied in a given context. Next we explain the basic constraints in order to create strategies.

### 5.2.2.1 Parameter Compatibility

The set of parameter vectors $\Pi$ is built over all possible parameter vectors according to parameter domains associated to basic tactics. Exchange a single parameter values, means the initial and resulting vector must be compatibles, i.e., the exchanged value must be in the same domain set, and the others values must remain unchanged.

**Definition 5.5** Two parameter vectors $\pi, \pi' \in \Pi$ are said to be compatible if

$$\pi, \pi' \in D_1 \times \ldots \times D_n \wedge \exists j \mid 1 \leq j \leq n, \pi_j \neq \pi'_j \wedge \forall i \mid 1 \leq i \leq n, i \neq j,\ \pi_i = \pi'_i$$

i.e., only one parameter value is changed. Thus, the constraint $\texttt{compatible}(\pi, \pi')$ checks if $\pi$ and $\pi'$ parameters vectors are compatible. ♢

#### 5.2.2.1.1 Time management

Note that one special case concerns *time parameters*, whose domains changes depending on the strategy context. These parameters are managed at a global level such that the available time is used optimally. There are two levels of available time: the total time available for the solver and the available time for a tactic explicitly defined by the function $\texttt{try-for}_{/2}$. Therefore, once the $\texttt{try-for}_{/2}$ values have been assigned, the remaining total time is distributed for the remaining sub-strategies. The same process is thus recursively applied to reach the whole strategy. For illustrate this, we define the Example 5.2.

**Example 5.2** Let $T_m$ be the strategy defined in Figure 5.2. If this strategy is used to solve a SMT formula instance with a global time-limit of 1 second, i.e., 1000 milliseconds, the time is distributed in tactics as follows:

```
1  (and-then
2           simplify
3           (try-for qe-sat 200)
4           (and-then
5                   qe
6                   (try-for smt 300)
7
8           )
9           smt
10 )
```

Figure 5.2 – Example of Time management in Strategy $T_m$.

1. Time is assigned to children of the strategy root. Thus, the solver tactic `qe-sat` has 200 millisecond available. Then 800 milliseconds remains to be assigned.
2. As no more tactics, in the same level, have defined a `try-for` function, the distribution continues in sub-strategies. In this case, the only sub-strategy is defined between lines 4 and 8.
3. The tactic `smt` (line 6), the unique element of the sub-strategy with `try-for` function defined, has 300 milliseconds available. Therefore, 500 milliseconds remains to be assigned.
4. Left unassigned tactics have not `try-for` function defined. Thus, the remaining time is assigned to them in order of instatiation, i.e., from top to down. However, as heuristics tactics proceeds in a negligible time, the solver `smt` (line 9) is assigned with the 500 millisecond left. ♢

#### 5.2.2.2 The Solver Constraint

Given the way generated strategies are applied (see Section 4.4.2) to solve SMT instances, some requirements must be fulfilled:

1. Strategies must be well-formed with regards to Z3 strategy language.
2. Solvers (tactics) are required in the strategy in a particular position, generally defined by the combinators functions used.

The first requirement is a basic restriction in order to generate correct strategies options to be evaluated. Meanwhile, the second requirement ensure the use of a tactical able to solve SMT instances. Note that, some syntactically correct strategies could not use solver tactics. However, they are semantically incomplete to solve an SMT formula, especially in our case where the default strategy is completely replaced. Therefore, at least one solver tactic must be present in the generated strategies.

**Definition 5.6** The *solver constraint* function, $\texttt{solver\_c}(st)$, checks if solver tactics are present in a correct position the strategy $st$, ensuring a syntactically and semantically correct strategy. Therefore, $\texttt{solver\_c}(st)$ can be expressed by the following condition:

$$
\begin{array}{lcl}
st(\epsilon) = \texttt{and-then}_{/n} & \Rightarrow & \texttt{solver\_c}(st|_n) \\
st(\epsilon) = \texttt{or-else}_{/n} & \Rightarrow & \forall i \in 1..n,\ \texttt{solver\_c}(st|_i) \\
st(\epsilon) = \texttt{try-for}_{/2} \vee st(\epsilon) = \texttt{using-params}_{/2} & \Rightarrow & \texttt{solver\_c}(st|_1) \\
st(\epsilon) \in \texttt{Probe} \cup \texttt{Heuristic} & \Rightarrow & \texttt{False} \\
st(\epsilon) \in \texttt{Solver} & \Rightarrow & \texttt{True}
\end{array}
$$

♢

To illustrate the previous constraint some cases of well-formed and ill-formed strategies are given in the following example, based in Figure 5.3.

**Example 5.3** Let $st_1$ and $st_2$ be two syntactically well-formed strategies shown in Figure 5.3a and 5.3b respectively. The solver constraint checks them as follows:

— In $st_1$, $\texttt{and-then}_{/4}$ function is used as root. To satisfy $\texttt{solver\_c}(st_1)$, the last tactic (line 9) inside this strategy must be a solver, but it is a composed tactic, sub-strategy $st_{1a}$, with $\texttt{or-else}_{/3}$ combinator as root. Then, $\texttt{solver\_c}(st_{1a})$ must be satisfied, i.e., each tactic in $st_{1a}$ must be a solver. Lines 10 and 11 show two sub-strategies, $st_{1a_1}$ and $st_{1a_2}$, in which $\texttt{try-for}_{/2}$ is applied to different solvers (`qe-sat` and `sat`). Therefore, $\texttt{solver\_c}(st_{1a_1})$ and $\texttt{solver\_c}(st_{1a_2})$ are evaluated to `True`. Moreover, last tactic (line 12) is a sub-strategy $st_{1a_3}$ uses as root the function $\texttt{and-then}_{/2}$. As $st_{1a_3}$ last tactic is a solver (`smt`), $\texttt{solver\_c}(st_{1a_3})$ is set to `True`. Consequently, $\texttt{solver\_c}(st_1)$ is satisfied.

— In $st_2$, $\texttt{or-else}_{/4}$ function is used as root. To satisfy $\texttt{solver\_c}(st_2)$, tactics in lines 2, 3, 4 and 12 must be solvers. Line 1 and 2 show two sub-strategies, $st_{2a}$ and $st_{2b}$, in which $\texttt{try-for}_{/2}$ function is applied to different solvers (`qe-sat` and `sat`). Therefore, $\texttt{solver\_c}(st_{2a})$ and $\texttt{solver\_c}(st_{2b})$ are evaluated to `True`. In line 4, a composed tactic $st_{2c}$ is used with $\texttt{and-then}_{/2}$ combinator as root, thus $\texttt{solver\_c}(st_{2c})$ must be satisfied. Last tactic in $st_{2c}$ (line 10) is another nested sub-strategy, $st_{2c_1}$, whose solver constraint must be also satisfied. As $st_{2c_1}$ strategy is another application of $\texttt{try-for}_{/2}$ in a solver (`smt`), both $\texttt{solver\_c}(st_{2c_1})$ and $\texttt{solver\_c}(st_{2c})$ are mapped to `True`. Last tactic (line 12) is a conjunctive sub-strategy, $st_{2d}$, using as root $\texttt{and-then}_{/3}$ combinator. As the last $st_{2d}$ tactic is the solver `smt` (line 15), $\texttt{solver\_c}(st_{2d})$ is set to `True`. Consequently, $\texttt{solver\_c}(st_2)$ is satisfied.

```
(and-then
        simplify
        propagate-values
        (using-params
                simplify
         :pull_cheap_ite true
         :local_ctx true
         :local_ctx_limit 10000000
        )
        (or-else
                (try-for qe-sat 1000)
                (try-for sat 1000)
                (and-then
                        qe
                        smt
                )
        )
)
```

(a) A well-formed strategy, $st_1$, using `and-then`$_{/4}$ function as root.

```
(or-else
        (try-for sat 100)
        (try-for qe-sat 500)
        (and-then
                (using-params
                        simplify
                 :pull_cheap_ite true
                 :local_ctx true
                 :local_ctx_limit 10000000
                )
                (try-for smt 500)
        )
        (and-then
                solve-eqs
                qe
                smt
        )
)
```

(b) A well-formed strategy, $st_2$, using `or-else`$_{/4}$ function as root.

```
(and-then
        simplify
        propagate-values
        (using-params
                simplify
         :pull_cheap_ite true
         :local_ctx true
         :local_ctx_limit 10000000
        )
        (or-else
                elim-uncnstr
                (try-for sat 1000)
                (and-then
                        qe
                        ctx-simplify
                )
        )
)
```

(c) An ill-formed strategy, $st'_1$, using `and-then`$_{/4}$ function as root.

```
(or-else
        (try-for sat 100)
        ctx-simplify
        (and-then
                (using-params
                        simplify
                 :pull_cheap_ite true
                 :local_ctx true
                 :local_ctx_limit 10000000
                )
                (try-for smt 500)
        )
        (and-then
                qe
                smt
                solve-eqs
        )
)
```

(d) An ill-formed strategy, $st'_2$, using `or-else`$_{/4}$ function as root.

Figure 5.3 – Examples of well-formed and ill-formed strategies with regards to `solver_c`$(st)$ constraint. Blue and red highlighted lines show tactics which satisfy and break, respectively, the `solver_c`$(st)$ constraint.

Accordingly, both strategies, $st_1$ and $st_2$ are well-formed with respect to the *solver constraint*. Now, we define the following structural modification rules:

$$r_1 : \delta(s, \pi) \rightarrow h$$

$$r_2 : s \rightarrow h$$

Let $st'_1$ (Figure 5.3c) and $st'_2$ (Figure 5.3d) be two strategies generated by modifying $st_1$ and $st_2$, respectively, by using rules $r_1$ and $r_2$ as follows:

$$st_1 \xrightarrow{r_1 \text{ (line 10)}} st_1^- \xrightarrow{r_2 \text{ (line 14)}} st'_1$$

$$st_2 \xrightarrow{r_1 \text{ (line 2)}} st_2^- \xrightarrow{r_2 \text{ (line 15)}} st'_2$$

The solver constraint check them as follows:

— In $st'_1$, $\texttt{solver\_c}(st'_1)$ is violated because some sub-strategy tactics are in a forbidden position. As explained for $st_1$ strategy, tactic (line 9) must be a solver, but it is a sub-strategy $st'_{1a}$, with $\texttt{or-else}_{/3}$ function as root. Then, each tactic in $st'_{1a}$ must be a solver. But, in line 10, rule $r_1$ introduced a heuristic (`elim-uncnstr`), mapping $\texttt{solver\_c}(st'_{1a})$, and by consequence $\texttt{solver\_c}(st'_1)$, to `False`. Same analysis could be done with `ctx-simplify` heuristic exchanged by using rule $r_2$ in line 14. This change breaks $\texttt{solver\_c}(st'_1)$ and $\texttt{solver\_c}(st'_{1a})$ by setting $\texttt{solver\_c}(st'_{1a_3})$ to `False`, because $\texttt{and-then}_{/2}$ combinator (line 12) needs a solver in that position.

— Strategy $st'_2$ have similar problems to those shown in $st'_1$. The violation of $\texttt{solver\_c}(st'_2)$ happens because of two components. In line 2, a solver must in that position, but `ctx-simplify` was introduced using rule $r_1$. Also, conjunctive strategy $st'_{2d}$ (line 12) should include a solver in its last position (line 15), to avoid global solver constrain breaking. However, $r_2$ rule inserted the heuristic `solve-eqs` in that place, thus $\texttt{solver\_c}(st'_{2d})$ is `False`.

Therefore, both modified strategies $st'_1$ and $st'_2$ are ill-formed with respect to the *solver constraint*. ♢

## 5.3 The Engine System

As rules defines how a strategy could be transformed, **the engine system** defines how to apply them. Thus, our automated strategy generation process is *an engine with various options applied to a given class of SMT problems which found suitable strategies generated by the use*

*of rules and improve SMT solver performance under certain solving conditions*, and it could be formalised as the following:

$$\texttt{Engine}[R, Is, Lspct, Ltopi, Ltb](Logic, Topi)$$

where, an `Engine` is indeed the main algorithm that applies rules for generating optimised strategies (w.r.t. to an evaluation function); it also provides the best generated strategy as output. The engine options (between square brackets [...]) are basic elements needed by the engine for its funtionability. Meanwhile, the solving conditions (indicated between round brackets (...)) are necessary values for validate obtained results.

### 5.3.1 The Engines

In order to efficiently explore the search space for building strategies (see Section 5.1.1), several local search algorithms are used as strategy generation engines based on evolutionary computing [17] and local search [19] techniques. Note that this type of algorithms have specific elements which allow us to consider several possible configurations of the strategy generation process.

**Algorithm 5.1:** Evolutionary Algorithm Scheme

**Input:** a SMT-LIB logic set of instances,
an initial strategy $Is$,
a population $size$,
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:** Optimised strategy $st^*$

1: Initialise *population* using $Is$
2: **repeat**
3: $r \leftarrow select_R(R)$
4: $ar(r) = n$
5: $Ind^n \leftarrow select_I(n, population, fitness)$
6: $Ind^n \rightarrow_r Ind'^{n'}$
7: *insert*( $Ind'^{n'}$,*population*,*fitness*)
8: $st^* \leftarrow$ best(*population*)
9: **until** end$_C$
10: **return** $st^*$

Thus, what we call an algorithmic engine is indeed a generic evolutionary process which is instantiated by different components, being the most important the rules set to be applied (see Section 5.2).

From now on, we use the classic vocabulary of evolutionary computation [17] to distinguish our different engines. Remind that our basic solutions or *individuals* are strategies corresponding to trees. Basically, Genetic Programming [40, 41] and its grammar-based derived [44, 48] aim at managing the structures of the trees, in particular applying crossover and mutation operators.

The mentioned generic evolutionary process is summarised in Algorithm 5.1. Here the classic evolution loop is applied on individuals of the population. At each iteration, an evolution rule is chosen. Then, the required number of individuals are selected and processed by the rules. The resulting individual(s) are then classically inserted in the population. In the following sections, we detail the components and setting of each algorithmic engine according to this general scheme.

### 5.3.2 Engine Options

We classify the engine settings as components options and learning parameters of a selected engines.

#### 5.3.2.1 Components Options

We call *components options* to the basic elements that define the strategy design process. We identify two main components options: the set of rules and the initial strategy.

1. **Rules (**$R$**)** is a set of rules that can be applied to strategies. These rules may modify the structure of a strategy, as well as its behavioral parameters (see Section 5.2). Basically, we consider variation rules to modify a strategy but we also consider recombinations of strategies. The *variation rules* correspond to mutation operators in evolutionary computation, while some variation operators also include recombination operators, i.e., exchange parts of strategies as classically performed in Genetic Programming [40, 41]. Here, variation rules aim at modifying a strategy by changing one of its elements and introducing possible new values (w.r.t. Z3 strategy language), while recombination rules only use existing strategies and values for creating new strategies.

   Therefore, based on the of rules classification previously described, we have four subsets of rules. Thus, as shown in Table 5.1, we consider a set of rules $R$ that is partitioned into structural variations rules ($SV$), structural recombination rules ($SR$), behavioral variation rules ($BV$), and behavioral recombination rules ($BR$).

2. **Initial Strategy (**$Is$**)** is a starting strategy (or a set of starting strategies). Let us remind that generating strategies from scratch is possible but rather intractable due to the size of the search space. Therefore, it is better (and even sometimes necessary) to get default strategies to initialise the process.

| Evolutionary operator / Rules ($R$) | Mutation | Crossover |
|---|---|---|
| Structural Rules ($S$) | Structural Variation ($SV$) | Structural Recombination ($SR$) |
| Behavioural Rules ($B$) | Behavioural Variation ($BV$) | Behavioural Recombination ($BR$) |

Table 5.1 – Rules classification: Categorizing by means of classic evolutionary operators.

#### 5.3.2.2 Learning Parameters

We call *learning parameters* the options values that define the computational aspects of the generation process. The engine system defines the following options:

— **Learning Sample Percentage** ($Lspct$) represents the size of subset from the complete given set of instances used to learn and generate strategies. Note that some cases, use the whole set of instances related to a SMT logic, could lead to an expensive computational and/or time resource process.

— **Learning Timeout per Instance** ($Ltopi$) is the timeout that is used within the learning process for evaluating the performance of the generated strategies for each instance of the learning set. Note that some real-life execution conditions, e.g. SMT-COMP 40 minutes per instance timeout, are impossible to use for generate strategies because generate an excessive time consumption.

— **Learning Time Budget** ($Ltb$) is the total amount of time allowed for generating a new strategy, i.e., an engine time-limit ending criteria.

### 5.3.3 Solving Conditions

Let us now define the validation phase by means of solving conditions applied over target that includes a set of instances and timeout conditions.

— **Logic** corresponds to a set of instances for a selected SMT logic. The logics considered in this work will be presented in Section 5.4.2.

— **Timeout per Instance** ($Topi$) is the time allowed for solving one instance of the given logic. Note that a generated strategy under certain timeout condition ($Ltopi$) could be used in several execution conditions. The problem introduced the difference of time between learning and execution will be discussed later on.

Figure 5.4 provides an overview of the whole process. The above-mentioned parameters and components are presented as well as their interactions with the algorithmic engine and the

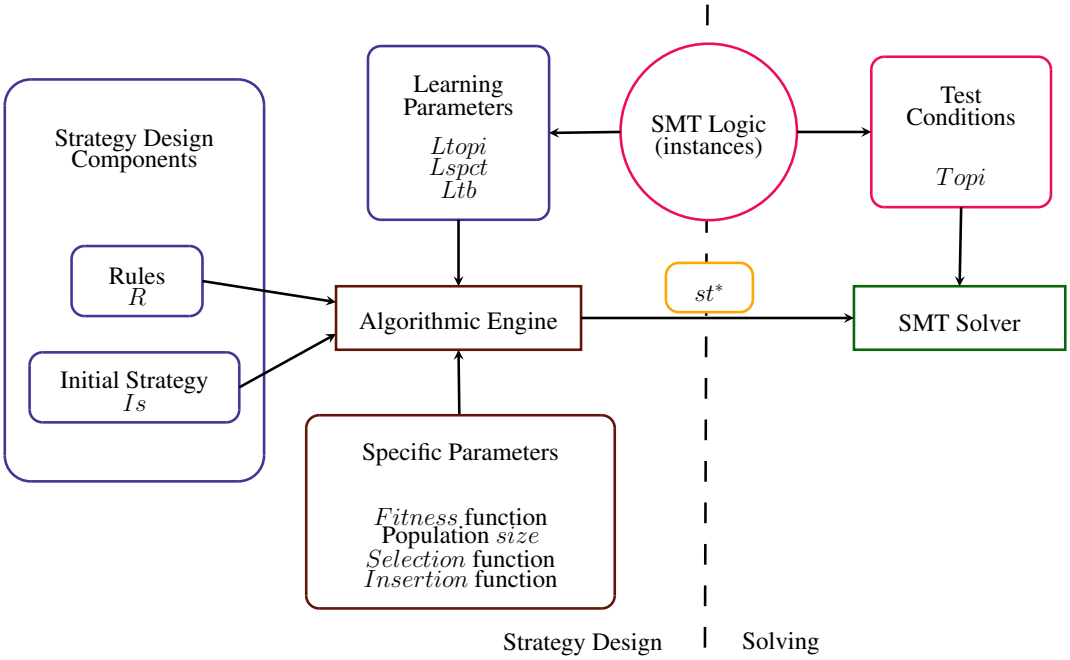



Figure 5.4 – Automated Generation of Strategies: methodology overview

solving process. On the left side of the figure, the learning phase is detailed. The right side of the figure describes the validation phase. This figure also allows us to clearly highlight the inputs (e.g., the options and the specific parameters) and outputs of the algorithmic engine. Note also some engine specific parameters that we do not detailed above: the fitness function to evaluate solving strategies, the size of the population used in the engine, the selection and insertion functions of individuals in the population. These specific parameters will be described later with regards to its corresponding engine.

## 5.4 Common Experimental Setup

Despite engines have several differences in their setup to perform, they also have also many common setup elements. Meanwhile, the particular configuration of each engine is explained in their corresponding chapters, we show up next the shared experimental configurations used by all designed engines.

### 5.4.1 Strategy Language

As shown in Section 4.3.1, three sets of constant terminal symbols, i.e., basic SMT components, are used to write strategies: `Probe`, `Heuristic`, and `Solver`. An explanation of these sets and the basic tactics that compose them, are given below:

— `Probe` set includes all property checkers for integer and real arithmetics (even those mixing both theories), their various domains (e.g., linear, non-linear, quantifier free), and the related theories (pseudo-Boolean, bit-vector) that could be obtained if some heuristics are applied. Model checkers are used to ensure that (un)satisfiability proofs are provided in the final result.

`Probe`={`is-pb`, `is-unbounded`, `is-qflia`, `is-qflra`, `is-qflira`, `is-ilp`, `is-qfnia`, `is-qfnra`, `is-nra`, `is-nia`, `is-nira`, `is-lia`, `is-lra`, `is-lira`, `is-qfbv`, `produce-proofs`, `produce-model`, `produce-unsat-core`}.

— `Heuristic` includes the most important techniques to propagate or reduce arithmetic formulas. It also contains a set of tactics that may transform arithmetics SMT equations into simple SAT instances.

`Heuristic`={`simplify`, `propagates-values`, `ctx-simplify`, `solve-eqs`, `elim-uncnstr`, `add-bounds`, `propagate-ineqs`, `normalize-bounds`, `eq`, `lia2pb`, `pb2bv`, `max-bv-sharing`, `bit-blast`, `aig`}.

— `Solver` includes the basic satisfiability verification techniques of Z3, as well as built-in solvers for the selected quantifier free logics.

`Solver`={`smt`, `sat`, `qe-sat`, `qflia`, `qflra`}.

These sets take into account most values used by the default Z3 strategies for linear arithmetic theories, including parameters vectors for behavioural engines. Let us also remark that if some initial strategy, used by an engine, has tactics not included in these sets, they are automatically incorporated to the corresponding set according to the Z3 documentation [180].

### 5.4.2 SMT Logics and Instances

For experiments, engines will focus on benchmarks issued from different logics from the SMT-LIB database. We focus here on different variants of Linear Arithmetic logics. `LRA` instances correspond to closed linear formulas in linear real arithmetic. The `QF_LRA` logic corresponds to quantifier free linear real arithmetic. These instances are boolean combinations of inequations between linear polynomials over real variables. `LIA` instances are boolean combinations of inequations between linear polynomials over integer variables. Again, `QF_LIA` are quantifier free formulas.

These benchmarks are also classified according to the expected output results: *known* and *unknown* instances. Known status corresponds thus to benchmarks whose satisfiability result was defined at that time, meanwhile unknown status refers to not assessed results. Actually, in SMT competitions, SMT solvers aim at classifying instances into *sat* or *unsat* instances when they succeed to solve them and *unknown* when they are not able to produce a result.

The selected instances were fully used in 2015 [165] and 2016 [164] SMT-COMP. In 2017 [163] and 2018 [162] SMT Competition, known and unknown instances were joined in a unique set. Also, some instances have also been deleted. Concerning LRA, more than 2000 instances were added.

| Logic | Instances | |
|---|---|---|
| | *known* | *unknown* |
| Linear Integer Arithmetic (`LIA`) | 210 | 189 |
| Linear Real Arithmetic (`LRA`) | 339 | 282 |
| Quantifier Free `LIA` (`QF_LIA`) | 5893 | 302 |
| Quantifier Free `LRA` (`QF_LRA`) | 1626 | 56 |

Table 5.2 – SMT-LIB Logics: Selected sets of instances characteristics.

Let us note that Linear Arithmetic logic family has many important SMT applications in several computer science topics, particularly Search-Based Software Engineering problems. Following the SMT-LIB classification, instances that have already been solved are called *known*, and the others *unknown*. Table 5.2 presents the characteristics of the different sets of benchmarks.

### 5.4.3 Learning Parameters

Learning parameters have also common settings for engines. All engines have the same maximum learning time budget ($Ltb$) of 2 days, which acts as one of the algorithm ending criteria. This value is defined to allow a correctly learning procedure. We discuss this value selection in the following Chapters. Meanwhile, the size of the learning sample ($Lspct$) depends on two engine elements:

— Type of rules used to generate strategies.

— Hardness to completely solve a selected logic subset.

Thus, as summarised in Table 5.3, the learning process is achieved on the whole instance (sub)set of logics if only *Structural Rules* are used, with exception of `QF_LIA` *known* and `QF_LRA` *known* subsets. And, if the subset is hard to tackle, i.e., is not completely solved using a engine generated strategy or is not easy learn from the entire set of instances, a 10% learning sample alternative is also used in order to reduce resource and time consumption by iteration. Last scenario also define $Lspct$ values when *Behavioural Rules* are introduced. The learning sample size corresponds to 10% of the instances set, if the same sample is the *only* option to learn by using Structural Rules. Otherwise, learning sample corresponds to all instances in the selected set.

| *Lspct* | | 100% set | | | |
|---|---|---|---|---|---|
| Rules | | $\{S\}$ | | $\{B, S \cup B\}$ | |
| Logics | LIA | ✓ | ✓ | ✓ | ✓ |
| | LRA | ✓ | ✓ | ✓ | ✓ |
| | QF_LIA | ✗ | ✓ | ✗ | ✓ |
| | QF_LRA | ✗ | ✓ | ✗ | ✓ |
| *Lspct* | | 10% set | | | |
| Rules | | $\{S\}$ | | $\{B, S \cup B\}$ | |
| Logics | LIA | ✗ | ✗ | ✗ | ✗ |
| | LRA | ✗ | ✓ | ✗ | ✗ |
| | QF_LIA | ✓ | ✓ | ✓ | ✗ |
| | QF_LRA | ✓ | ✗ | ✓ | ✗ |
| Subset | | *known* | *unknown* | *known* | *unknown* |

Table 5.3 – Learning Samples: Classifying by hardness of the selected instance set.

### 5.4.4 Time Conditions for Validation and Learning

We explain both time limit per instance options together, for learning and solving stages, because they are intrinsically related. Of course, we consider as timeout per instance ($Topi$) the amount used in the SMT-COMP from 2015 [162, 165] to solve instances: 40 minutes (2400 seconds). However, this value is too big to use it as learning timeout per instance ($Ltopi$), e.g., in LRA unknown, worst scenario it could take around seven days to solve the whole set of 282 instance. If we consider Z3 performance in 2016 SMT COMP, were 36 instances remained unsolved, address the whole set would take around one day, allowing preform at most two iterations per engine. Therefore, $Ltopi$ for engines use reduced values that allow to perform several iterations: 1 second or 10 seconds.

| $Ltopi$ \ $Topi$ | $1[s]$ | $10[s]$ | $2400[s]$ |
|---|---|---|---|
| $1[s]$ | ✓ | ✗ | ✗ |
| $10[s]$ | ✗ | ✓ | ✓ |
| $2400[s]$ | ✗ | ✗ | ✗ |

Table 5.4 – Timeout per instance settings for learning and solving phases.

Thus, engines learn in a reduced timeout per instance budget to be then evaluated in SMT-COMP conditions. Table 5.4 summarises the learning timeout per instance ($Ltopi$) used for the generation strategies, and the timeout per instances ($Topi$) used for validate these strategies according to the learning process.

### 5.4.5 Fitness Function for Strategies

We evaluate strategy effectiveness through a *fitness function* over Z3 solver. This function involves the number of solved instances and, as a second criterion, the time used for solving these instances by using a selected strategy. We define our fitness function as:

$$f : PStrat \mapsto \mathbb{N} \times \mathbb{N}$$
$$f(st) = (i(st), t(st))$$

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 $st, st' \in \texttt{Strat}, f(st) = (i, t), f(st') = (i', t')$, we have $f(st) \succ f(st')$ if and only if $i > i'$, or $i = i'$ and $t < t'$.

As previously mentioned, some changes could alter the scope of the strategy, i.e., its ability to solve all the instances of a logic. Our fitness function aims at evaluating the solving performances on the learning set and do not explicitly address the generality of the strategy. Nevertheless, due to the learning process and the instance sets, generated strategies are efficient in terms of solved instances. We focus indeed on specific strategies improvement, guided by SMT competitions.

### 5.4.6 Workstation and Implementation

All our engines rely on a set of C programs which interact with Z3. Computations are performed on cluster *Taurus2* in **LERIA** (*Laboratoire d'Etude et de Recherche en Informatique d'Angers*) in the *Université d'Angers*. The cluster have the following features:

— Processors: 18 × Intel(R) Xeon(R) E5-2670, 2.8 Ghz, 10 cores.

— Nodes: 7 available, with 20 threads each one, and support for 50 simultaneous job execution.

— RAM: 63 GB RAM per node.

Engines were generated using the gcc compiler version 4.8.2 and Z3 theorem prover stable version 4.4.0.

## 5.5 Conclusions

In this chapter we formalise a scheme for automated generation of strategies. This framework is based in two core elements, a rules approach and an engine system.

The rules approach allows modify strategies using a set of rules based in evolutionary computing operators. We introduce the notation for defining rules, how they perform over a known strategy, and the set of constraints to be satisfied in order to generate well-formed strategies.

The engine system formalise the use of algorithms to apply the rules and sets the needed configuration parameters for their performance. We detailed the architecture of the system, and how the learning phase is related with the solving or validation stage.

Then, common settings for initialise performance of all engines was explained. Finally, the computational context of the implementation of each algorithm subscribed to the developed strategies generation framework was described.

# 6

# Simple Evolutionary Programming Engines

In this chapter, based on our Evolutionary Algorithm Scheme (see Algorithm 5.1 in Section 5.3), we present two engines inspired by previous works of evolutionary programming and parameter tuning [91, 92]. Each algorithm or engine focuses on a different type of modifying rules, i.e., Structural and Behavioural sets of rules, respectively. We show the impact produced by these types of rules on the generation strategy process, and consequently, on the performance of Z3 solver. We also decide which kind of rules will lead to the construction of more sophisticated engines.

## 6.1 Evolutionary Strategy Generator, StratGEN

Our first evolutionary programming engine is called, *Evolutionary Strategy Generator* or **StratGEN**. It modifies a simple strategy, which contains a fixed amount of heuristics and solving tactics, using mostly mutation-based structural rules. The specific setting components of this algorithm are explained in detail below.

### 6.1.1 Initial Strategy, $Is$

The starting point in StratGEN is a fixed skeleton strategy inspired by recurrent structures and tactics present in Z3 default strategies for the given SMT logic (see Section 5.4.2).

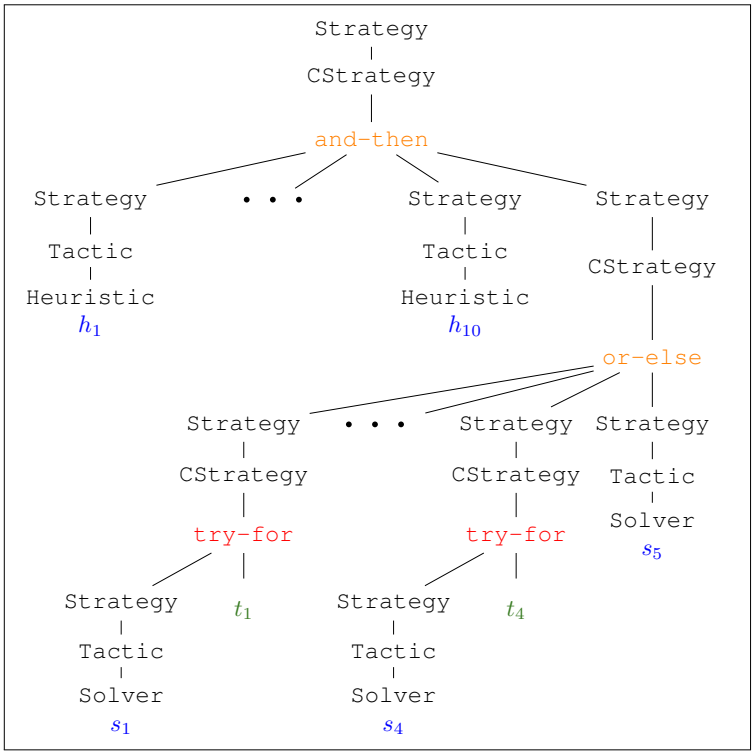



Figure 6.1 – StratGEN initial strategy structure: a fixed skeleton, defining relation between heuristics and solving tactics. It can be generated by using $\mathcal{G}_{Z3}$ derivation tree.

As shown in Figure 6.1, the main structure of the strategy is the conjunctive union of two defined segments based on the $\texttt{and-then}_{/11}$ function:

— A set of ten heuristics tactics:

$$h_i \in \texttt{Heuristic}; i \in \{1, \ldots, 10\}$$

These heuristics help to reduce and/or propagate original SMT formula before starting the solving process. They are applied sequentially one after the other.

— A disjunctive sub-strategy (using the $\texttt{or-else}_{/5}$ combinator) of five solving tactics and their corresponding time limit (set by the $\texttt{try-for}_{/2}$ function):

$$s_j \in \texttt{Solver}; j \in \{1, \ldots, 5\}$$

$$t_j \in \mathbb{N}; j \in \{1, \ldots, 5\}$$

As solvers are disjunctively related, if one fails it is discarded and the next one tries to tackle the SMT formula. This procedure is repeated until a solution to the formula (`sat` or `unsat`), or the final result is not determined (`unknown` or `timeout`). Note that only the last solver tactic has not a specific timeout parameter. This is because it uses all the

remaining time available with regards to the global time constraint, thus there is no need to specify its time limit.

#### 6.1.1.1 Available Strategy Language

The mentioned structure uses elements from sets of commonly-used tactics in order to generate the initial strategy for StratGEN. These sets are:

— $\texttt{Heuristic}_{\text{SGEN}} \subset \texttt{Heuristic}$, it contains ten eligible heuristics.

— $\texttt{Solver}_{\text{SGEN}} \subset \texttt{Solver}$, it is composed by seven common solving tactics.

— $\Pi_{\text{SGEN}} \subset \Pi$, it represents all timeout parameter values for solving tactics.

| | Tactics | | | | | |
|---|---|---|---|---|---|---|
| $\texttt{Heuristic}_{\text{SGEN}}$ | 1 | simplify | 2 | $\texttt{simplify}_{mod}$ | 3 | ctx-simplify |
| | 4 | $\texttt{ctx-simplify}_{mod}$ | 5 | solve-eqs | 6 | elim-uncnstr |
| | 7 | propagate-ineqs | 8 | split-clause | 9 | lia2sat |
| | 10 | skip | | | | |
| $\texttt{Solver}_{\text{SGEN}}$ | 1 | qe-smt | 2 | sat | 3 | smt |
| | 4 | qe-sat | 5 | qflia | 6 | qflra |
| | 7 | fail | | | | |

Table 6.1 – StratGEN Initial strategies: Heuristics and solvers tactic values.

Table 6.1 summarises the elements of $\texttt{Heuristic}_{\text{SGEN}}$ and $\texttt{Solver}_{\text{SGEN}}$ respectively. Despite of using common tactics and parameter values, it is necessary to explain the following:

— In heuristics values, $\texttt{simplify}_{mod}$ and $\texttt{ctx-simplify}_{mod}$ are modified versions of simplify and ctx-simplify respectively. Both tactics modify a set of their own default parameters values. Also, lia2sat is a composed tactic that joins a set of heuristics by a conjunction. It is recurrently used in z3 default strategies for linear arithmetic theories. Finally, skip is a void tactic.

— In solvers values, qe-smt is a tactic solver composed of heuristic qe and solver smt. Tactics qflia and qflra are z3 built-in solvers for QF_LIA and QF_LRA respectively. We selected them in order to check if they could be useful in the strategies generated for quantified linear arithmetic logics (LIA and LRA respectively). Finally, fail is a unknown result inducing tactic. It is useful as solver deleter in disjunction scenarios.

— Time-out parameters values in $\Pi_{SGEN}$ must respect instance global timeout ($Ltopi$) constraint.

#### 6.1.1.2 Initial strategy building

To generate the initial strategy using the mentioned structure and available components, some constraints must be defined:

1. No tactic is repeated. Two tactics with distinct parameter vectors, are considered as different elements.
2. Sub-strategies, i.e., composed tactics, are considered as single tactic. But, its components can not be repeated inside itself.
3. Time parameter value will be equivalent for all solvers with regards to the learning timeout per instance ($Ltopi$) limit.
4. Last solver will be `qflia` for `LIA` or `QF_LIA` logics, and `qflra` for `LRA` or `QF_LRA` logics.

Then, heuristics and solver tactics are randomly chosen and placed in the structure. To illustrate an $Is$ designed for StratGEN, we introduce the Example 6.1.

```
(and-then
        skip
        simplify
        (using-params
                simplify
         :pull_cheap_ite true
         :local_ctx true
         :local_ctx_limit 10000000
        )
        ctx-simplify
        (using-params ctx-simplify :max_depth 30 :max_steps 5000000)
        solve-eqs
        elim-uncnstr
        (and-then
                normalize-bounds
                lia2pb
                pb2bv
                max-bv-sharing
                bit-blast
                aig
        )
        propagate-ineqs
        split-clause
        (or-else
                (try-for sat 200)
                (try-for smt 200)
                (try-for qe-sat 200)
                (try-for
                        (and-then
                                qe
                                smt
                        )
                200)
                qflia
        )

)
```

Figure 6.2 – StratGEN initial strategy example, $Is_{LIA}$: Configuration used as $Is$ for `LIA` or `QF_LIA` logics, with $Ltopi = 1$ second.

**Example 6.1** Let $Is_{LIA}$ the initial strategy used by StratGEN for `LIA` and `QF_LIA` logics with a learning timeout per instance ($Ltopi$) of 1 second, as shown in Figure 6.2. Let us remark the following aspects:

— The tactical `simplify`$_{mod}$ is defined, between lines 4 and 9, as `simplify` (line 3) with a different parameter vector $\pi_1' \in \Pi$.

— The tactical `ctx-simplify`$_{mod}$ is defined, in line 11, as `ctx-simplify` (line 10) with a different parameter vector $\pi_2' \in \Pi$.

— The composed heuristic `lia2sat` is shown between lines 14 and 21. It is a conjunctive sub-strategy, generated by `and-then`$_{/6}$, which contains no repeated heuristics.

— The composed solver `qe-smt` is shown between lines 29 and 32. It is a `smt` based tactic which applies `qe` before starting the solving procedure.

— The last solver in the strategy (line 34) is `qflia`, because the strategy is designed to solve `QF_LIA` related logics.

— Time-out configuration is equal in each solver (lines 25 to 34), i.e., each binary function `try-for`$_{/2}$ has the same time parameter value: `200` milliseconds. Note that the last solver does not need an explicit `try-for`$_{/2}$ declaration because its execution time is bounded by the remaining time $Ltopi - 800 = 200$ milliseconds. ♢

### 6.1.2 Rules

In order to exchange initial strategy basic tactics, StratGEN relies on a set of rules, denoted $R_{SGEN}$. This set is mostly composed by mutation-based rules, i.e., Structural Variation rules for exchanging strategy tactics components and Behavioural Variation rules are included for time-handling. However, a set of Structural Recombinations rule is defined. Later we discuss why we still classify StratGEN as an Evolutionary programming algorithm despite it is using a crossover-based rule.

#### 6.1.2.1 Structural Variation Rules

Let $SV_{SGEN} \subset R_{SGEN}$ be the set of Structural Variation rules which modify tactical core components of a strategy in StratGEN. This set is composed of two generic rules:

— Modify Heuristic: allows to change a heuristic tactic $h$ by another $h'$ at a selected position. Let us remark that $s, s' \in \texttt{Heuristic}_{SGEN} \subset \texttt{Heuristic}$.

$$MH_{SGEN} : h \rightarrow h'$$

— Modify Exchange: allows to change a solver tactic $s$ by another $s'$ at a selected position. Let us remark that $s, s' \in \texttt{Solver}_{SGEN} \subset \texttt{Solver}$.

$$MS_{SGEN} : s \rightarrow s'$$

#### 6.1.2.2 Behavioural Variation Rules

Let $BV_{SGEN} \subset R_{SGEN}$ be the set of Behavioural Variation rules which exchange timeout values for solvers in a strategy with StratGEN. Note that time limit constitutes an important parameter for improving strategies, the management of the time allocated to the different tactics is handled carefully, given the global learning timeout per instance $Ltopi$.

— Modify Time-out: allows to change a solver timeout value. Thus, let $t$ be the $k^{\text{th}}$ component of the parameter vector $\pi$ of solver $s$ (denoted as $\pi_k = t$) which corresponds to the solver time limit in a strategy of StratGEN.

$$MT_{SGEN} : \texttt{try-for}_{/2}(s, t) \rightarrow \texttt{try-for}_{/2}(s, t')$$

Note that the parameter vector $\pi$ is turned into $\pi'$, where $\texttt{compatible}(\pi, \pi')$ is fulfilled because $t \neq t'$, i.e, the unique difference between $\pi$ and $\pi'$ is $\pi_k \neq \pi'_k$.

#### 6.1.2.3 Structural Recombination Rule

Let $SR_{SGEN} \subset R_{SGEN}$ be the set of Structural Recombination rules which generate a new offspring from a set of strategies, i.e., a population. This set is composed of the following rules:

— Uniform recombination: inspired by Wheel-Selection Crossover [31] evolutionary operator, this rule generates a new strategy using tactics present in the population at some specific position.

$$UR_{SGEN} : st^n \rightarrow \varepsilon(st_i|_1, \ldots, st_i|_{|Pos(st)|}) = st'$$

Note that $\varepsilon : \overbrace{\texttt{Strat} \times \ldots \times \texttt{Strat}}^{n \text{ times}} \rightarrow \texttt{Strat}$, uses a set of randomly selected function symbols $st_i|_p$ in the new strategy $st'$, i.e., $\varepsilon$ is such that $\forall p \mid 1 \leq p \leq |Pos(st)|,\ \exists i \mid 1 \leq i \leq n, st'|_p = st_i|_p$.

Although using a crossover-based rule, we still classifying StratGEN as an Evolutionary Programming engine. This is because the $UR_{SGEN}$ operator could be seen as an union of mutation-based variation operators given the fixed structure of StratGEN strategies, as shown in Example 6.2

**Example 6.2** Let $Is$ be the initial strategy such that $Is|_p \in \texttt{Heuristic}$ with $p \in Pos(st)$, then $\forall st \in \texttt{Strat}$ generated by StratGEN, $st|_p \in \texttt{Heuristic}$. Thus, if $\texttt{Heuristic}_p$ is the set of all heuristics present at position $p$ in a set of strategies (i.e., population), each subterm $st'|_p$ of an offspring generated by $UR_{SGEN}$ can be obtained by the following variation rule:

$$MH_{SGEN} : h \rightarrow h'$$

where $h, h' \in \texttt{Heuristic}_p \subseteq \texttt{Heuristic}_{SGEN}$. Note this could be extended for `Solver` tactics and timeout parameter configurations. ♢

### 6.1.3 Strategy encoding

Since the strategies treated by StratGEN have a fixed structure and only the *leaves* of the term grammar tree are modified, the solution could be represented as an ordered array containing real values, as shown in Figure 6.3.

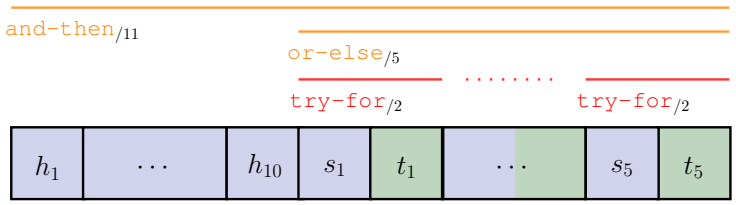



Figure 6.3 – StratGEN strategy encoding: An ordered array of real values representing exchangeable strategy values (leaves of $\mathcal{G}_{Z3}$ term grammar).

Note that a simple mapping is done for representing elements of $\texttt{Heuristic}_{SGEN}$ and $\texttt{Solver}_{SGEN}$ using values shown in Table 6.1. Meanwhile, time values must ensure not to exceeding the global learning timeout per instance ($Ltopi$). To address this problem, we map time values into real values between 0 and 1, allowing to correctly handle time distribution through proportion.

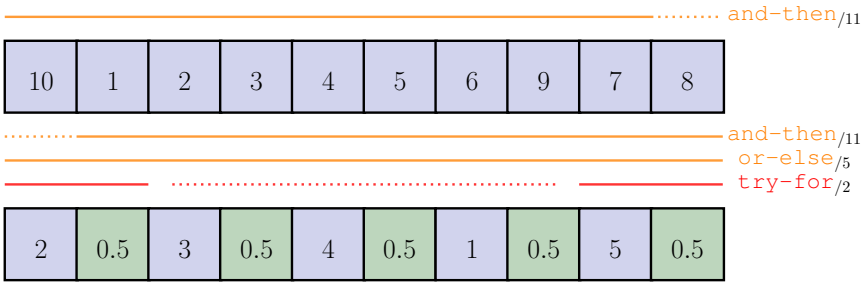



Figure 6.4 – StratGEN initial strategy encoded: Configuration used as $Is$ for `LIA` or `QF_LIA` logics, with $Ltopi$ of 1 second.

**Example 6.3** Let $A$ be an ordered array of real values as shown in Figure 6.4 for representing $Is_{LIA}$ of Example 6.1. Note we could read the array $A$ from left to right to compose the represented strategy given its fixed structure.

Thus, we know the first heuristic component (line 2 in Figure 6.2) is labeled with value 10 that encodes the `skip` tactic (see Table 6.1). We can continue until the last heuristic (line 23), which is tagged with the value 8 and it represents the `split-clause` tactic.

The same procedure applies to the solvers, where the first (line 25) is encoded as number 2 matching with `sat` value, and the last solver (line 34) is labeled with value 5 corresponding to `qflia` tactic.

Moreover, each solver execution time limit have the same treatment. However, its values are obtained as the proportion with regards to $Ltopi$ according to the $[0, 1]$ values set in the array $A$. Let $t^{[0,1]}$ be the mapping in the range $[0, 1]$ of a solver time limit $t$. As each solver in $Is_{LIA}$ has the same $t^{[0,1]}$ value, the corresponding time (in milliseconds) is obtained by the proportion formula:

$$Ltopi \times \frac{t_1^{[0,1]}}{\sum_{i=1}^{5} t_i^{[0,1]}} = 1000 \times \frac{0.5}{5 \times 0.5} = \frac{1000}{5} = 200 \qquad \diamondsuit$$

### 6.1.4 Population

Once defined the representation of a strategy as an ordered array of real values, we could consider this strategy mapping as an *individual* of the StratGEN engine. Thus, the population is a set of fixed structure strategies mapped as arrays. The *population size* parameter is automatically configured as the maximum number of possible candidate values that a discrete strategy component could have. In StratGEN, heuristics are the strategy component with biggest cardinality in its domain set (given by $\texttt{Heuristic}_{SGEN}$), thus population size is set to 10.

The initial population is set up by assigning ordered values of its domain to all parameters in each individual. For continuous domains, an initial discretisation must be performed. Thus, timeout proportion values (range $[0, 1]$) are reduced to set $\{0.1, 0.2, \ldots, 0.9, 1.0\}$.

Then, as shown in Algorithm 5.1, StratGEN population can evolve by using rules and individuals chosen by *selection* functions, and whose results are included to the population by means of a fitness-based *insertion* function.

#### 6.1.4.1 Selection functions

Selection functions are in charge of choosing (at some point of the evolution process, i.e., iteration) a rule from $R_{SGEN}$ and a set of individuals to be used with it.

Algorithm 6.1, shows how the $select_R$ function works in StratGEN. It starts selecting a uniform recombination rule ($r_r \in SR_{SGEN}$), and then, in next generation, it picks a structural or behavioural variation rule ($r_v \in SV_{SGEN} \cup BV_{SGEN}$).

Meanwhile, the $select_I$ function picks the whole population as the set of individuals for applying a rule if the rule arity is equal to the population size, otherwise the last inserted population member (see Algorithm 6.2). Note that the arity of any specific rule $r_r \in BR_{SGEN}$ is $ar(r_r) = size$, then uniform recombination rules will be performed over the whole population.

**Algorithm 6.1:** StratGEN $select_R$ function

**Input:** a set of rules $R_{SGEN}$
**Output:** A rule $r$
1: **if** iteration number odd **then**
2: $r \leftarrow$ `random`$(SR_{SGEN})$
3: **else**
4: $r \leftarrow$ `random`$(SV_{SGEN} \cup BV_{SGEN})$
5: **end if**
6: **return** $r$

**Algorithm 6.2:** StratGEN $select_I$ function

**Input:** an amount of individuals, $n$
a population, *population*
a fitness function *fitness*
**Output:** a set of individual $ind^n$
1: **if** $n = size$ **then**
2: $ind^n \leftarrow$ *population*
3: **else**
4: $ind^n \leftarrow$ `last_inserted`(*population*)
5: **end if**
6: **return** $ind^n$

Consequently, the arity of $r_v \in SV_{SGEN} \cup BV_{SGEN}$ is $ar(r_s) = 1$, hence behavioural rules will be performed over the last inserted member of the population, which corresponds to the offspring of the last $r_r$ rule selected as shown below.

#### 6.1.4.2 Insertion function

Once a new individual, i.e., a strategy, is generated, we need to insert it in the population. StratGEN insertion function defines if a new individual will be part of the current population by means of a given *fitness* function (see Section 5.4.5).

Algorithm 6.3 explains how *insert* function works in StratGEN. The offspring of a $r_r \in SR_{SGEN}$ rule replaces the worst individual of the population. The result of a variation rule $r_v \in SV_{SGEN} \cup BV_{SGEN}$ replaces worst element of the population (different from the last $r_r$ offspring inserted) if its fitness is better. Note that a complete complete cycle is done each two generations, i.e., first a new strategy is generated using $r_r \in SR_{SGEN}$, to then modify with a variation rule $r_v \in SV_{SGEN} \cup BV_{SGEN}$.

**Algorithm 6.3:** StratGEN *insert* function

**input:** an individual set, $Ind'^{n'}$
a population, *population*
a fitness function *fitness*
**output:** a new *population* generation
1: $new \leftarrow$ `best`($Ind'^{n'}$)
2: **if** iteration number odd **then**
3: $worst \leftarrow$ `worst`(*population*,*fitness*)
4: *population* $\leftarrow$ `replace`($worst$,$new$)
5: **else**
6: *last* $\leftarrow$ `last_inserted`(*population*)
7: $worst \leftarrow$ `worst`(*population-last*,*fitness*)
8: *population* $\leftarrow$ `replace`($worst$,$new$,*fitness*)
9: **end if**
10: **return** *population*

## 6.2 Evolutionary Strategy Tuner, StraTUNE

Our second evolutionary programming algorithm is called *Evolutionary Strategy Tuner* or **StraTUNE**. It aims at setting automatically the parameter configuration vector of tactics in a strategy through the uses of mutation-driven rules. In this scenario, strategy structure and tactics values remain fixed during the whole evolution procedure. The specific engine components are explained below.

### 6.2.1 Initial Strategy, $Is$

As StraTUNE looks to improve strategies by changing their parameter vector configuration, it does not need to build or modify the structure, but to calibrate the behaviour of a strategy. Thus, StraTUNE uses $Is$, the Z3 default strategy for a given logic, as the initial strategy.

Note that Z3 default strategies have several nuances depending on the related type of logic. As shown in Figure 6.5, `LIA` and `LRA` default strategies (see Figure 6.5a) have a more complex structure, tactics, and parameter vectors than `QF_LRA` default strategy (see Figure 6.5b) which is composed of a single tactic with a specific parameter vector.

### 6.2.2 Rules

In order to calibrate strategy performance for a set of instances of a selected logic, a set of rules (denoted as $R_{STUNE}$) is defined for modifying parameter vector values. This set is only composed of Behavioural Rules, where mutation-based rules, i.e., variation rules, predominate. However, as in StratGEN, a recombination rule set is defined. Despite of this, we still clas-

```
(and-then
        simplify
        propagate-values
        (using-params
                ctx-simplify
         :max_depth 30
         :max_steps 5000000
        )
        (using-params
                simplify
         :pull_cheap_ite true
         :local_ctx true
         :local_ctx_limit 10000000
        )
        solve-eqs
        elim-uncnstr
        simplify
        (or-else
                (try-for smt 100)
                (try-for qe-sat 1000)
                (try-for smt 1000)
                (and-then qe smt)
        )
)
```

(a) Strategy for `LIA` and `LRA` logics.

```
(using-params
        smt
 :arith.greatest_error_pivot true
)
```

(b) Strategy for `QF_LRA` logic.

Figure 6.5 – Z3 default strategies

sify StraTUNE as an evolutionary programming engine because of the mutation nature of the recombination rule set as explained later.

Note also that strategies involve many parameters. Indeed, most tactics have particular configurations that determine their behaviors. When a tactic is involved several times in a strategy, each copy/clone is managed independently (using its own parameter vector). Combinators functions ($\Gamma$ set symbols) also manage a global vector of parameters, which is the concatenation of all parameters used in the sub-strategies. The total number of parameters may thus increase significantly with regards to the size of the strategy.

When a tactic is used, default parameters values are invoked: they may not be explicitly present in the strategy and thus, constitute implicit parameters. If a parameter value is explicitly given (i.e., explicitly written) in the strategy, it overwrites the default value and becomes thus an explicit parameter. This value appears in the strategy within `using-params` or `try-for` functions.

Thus, if a parameter is explicitly defined in the strategy, we introduce a rule to change its value according to its domain. We leave aside implicitly defined parameters in order to keep a reasonable search space, i.e., we do not introduce new explicit parameters into the strategy. Thus, let $EW$ be the set of components of parameter vector $\pi^{Is}$ explicitly written in the initial strategy $Is$. Moreover, we still manage carefully tactics timeout limits as in StratGEN (see Section 6.1.3).

#### 6.2.2.1 Behavioural Variation Rules

Let $BV_{STUNE} \subset R_{STUNE}$ be the set of Behavioural Variation rules which exchange parameter vector values of a strategy in StraTUNE. This set is composed of the following generic rule:

— Modify Parameter: allows to change the value of the $k^{th}$ component of a tactic parameter vector, $\pi_k \in EW$.

$$MP_{STUNE}: \ \delta(st, \pi) \rightarrow \delta(st, \pi')\{\texttt{compatible}(\pi, \pi') \wedge \pi_k \neq \pi'_k\}$$

Note this modification implies $\pi'_k \in EW$.

#### 6.2.2.2 Behavioural Recombination Rules

Let $BR_{STUNE} \subset R_{STUNE}$ be the set of Behavioural Recombination rules which generates a new strategy parameter vector from a set of same structure strategies, i.e., StraTUNE population. This set is formed by the following rule:

— Uniform recombination: as well as in StratGEN, based in Wheel-Selection Crossover [31], this generic rule generates a new parameter configuration vector based on the current settings of each individual in the population. As $st^n$ is a set of $n$ strategies, then $\pi^n = \pi^1, \ldots, \pi^n$ is the set of parameter vectors of $st^n$ , where $\pi^i$ is the parameter vector of the $i^{th}$ strategy.

$$UR_{STUNE} : st^n[\pi^n] \rightarrow st[\upsilon(\pi^1, \cdots, \pi^n)] = st[\pi']$$

where $\upsilon : \overbrace{\Pi \times \cdots \times \Pi}^{n \text{ times}} \rightarrow \Pi$ uses a set of randomly components $\pi^i_k$ for the new parameter vector $\pi'$, i.e., $\upsilon$ is such that $\forall k \mid 1 \leq k \leq |\pi|, \ \exists i \mid 1 \leq i \leq n, \pi'_k = {\pi^i}_k$.

Although we have a crossover-based rule, we still classify StraTUNE as an Evolutionary Programming engine. This is because the $UR_{STUNE}$ operator could be explained as the simultaneous application of $BV_{STUNE}$ rules, as shown in Example 6.4. Let us remark that when we change parameter vector of a set of strategies their structure remains unchanged.

**Example 6.4** Let $Is$ be the initial strategy and $\pi^{Is}$ its parameter vector such that its $k^{th}$ component $\pi^{Is}_k \in EW$. Then, $\forall st \in \texttt{Strat}$ generated by StraTUNE, $\pi_k \in EW$. Thus, if $\Pi_k$ is the set of the $k^{th}$ components in all parameters configuration ($\pi^n$) over the population ($st^n$), the variation rule:

$$MP_{STUNE}: \ \delta(st, \pi) \rightarrow \delta(st, \pi')\{\texttt{compatible}(\pi, \pi') \wedge \pi_k \neq \pi'_k\}$$

can be used to obtain each parameter value $\pi'_k$ of a offspring generated by $UR_{STUNE}$, where $\pi_k, \pi'_k \in \Pi_k$. ♢

### 6.2.3 Solution Encoding

Since the strategies treated by StraTUNE have the same structure and tactics components in all engine generations, we could represent each strategy parameter vector as an ordered array of real values. Note that the size of this array depends on the amount of explicit parameters defined in the initial strategy $Is$.

Note that most parameters values found in Z3 documentation [180] are easily expressed in real domain, but some must be mapped to this domain, e.g. boolean or categorical values. As well as in StratGEN, we use a mapping in the $[0, 1]$ range to proportionally distribute time values ensuring not to exceed global timeout per instance ($Ltopi$), including the last solver which exceptionally does not have an explicitly parameter value.

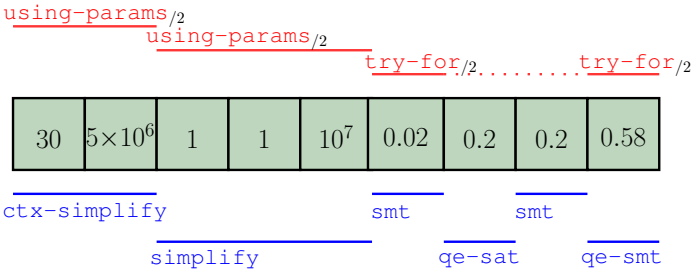



Figure 6.6 – StraTUNE initial strategy encoded: Configuration used as $Is$ for `LIA` or `LRA` logics, with $Ltopi$ of 5 second.

**Example 6.5** Let $Is$ be (see in Figure 6.5a) the initial strategy for StraTUNE for `LIA` logic with $Ltopi = 5$ seconds. Then, $A$ (Figure 6.6) is the array representing the configuration of explicitly written vectors in $Is$.

The first two slots of array $A$ corresponds to the two explicit parameters, `max_depth` and `max_steps` (line 6 and 7), associated to heuristic `ctx-simplify` (line 5). Then, we have three explicit parameters of tactic `simplify` (lines 9 to 14): `pull_cheap_ite`, `local_ctx` and `local_ctx_limit`. Note the first two parameters are truth values mapped to classic integer set $\{0 = \texttt{False}, 1 = \texttt{True}\}$. Last four array members represent time limits of $Is$ solvers (line 19 to 22), these values follow the same mapping principle as in StratGEN. For example, the second `smt` instantation (line 21) timeout is mapped to the real value $0.2$. This value is the solver timeout proportion that respects all other solver values and $Ltopi$:

$$Ltopi \times \frac{t_3^{[0,1]}}{\sum_{i=1}^{4} t_i^{[0,1]}} = 5000 \times \frac{0.2}{0.02 + 0.2 + 0.2 + 0.58} = \frac{5000}{5} = 1000 \qquad \diamondsuit$$

### 6.2.4 Population

Once parameter vector representation is defined for $Is$, we can use it through the entire evolutive process of StraTUNE. We use a specific real-value array as *individual* of the engine. Then, the population is set to individuals representing several parameter configurations for the same strategy. The *size of the population* is defined as the maximum number of possible values that a parameter could have. Of course, we cannot consider parameter values whose domain are continuous: a discretisation is performed by dividing the domain of this parameter in five equiproportional parts of the maximum value. The exception to this is the discretisation process for solver real range for timeout proportion, which is the same as explained in StratGEN (see Section 6.1). Thus, depending on the selected SMT logic, StraTUNE population size moves between ten and twelve individuals.

Then, as shown in Algorithm 5.1, StraTUNE population evolves by means of the rules and individuals chosen by *selection* functions. New population members are also included into the current population using a fitness-based *insertion* function.

#### 6.2.4.1 Selection functions

Selection functions are in charge of choosing, at some point of the evolution process (i.e., iteration), a rule from $R_{STUNE}$ and a set of individuals to generate a new parameter vector configuration.

**Algorithm 6.4:** StraTUNE *select*$_R$ function

**Input:** a set of rules $R_{STUNE}$
**Output:** A rule $r$

```
1: if iteration number odd then
2:     r ← random(BR_TUNE)
3: else
4:     r ← random(BV_TUNE)
5: end if
6: return r
```

As shown in Algorithm 6.4, the *select*$_R$ function in StraTUNE chooses at first a uniform recombination rule ($r_r \in BR_{STUNE}$), and then, in the next generation, it picks a behavioural variation rule ($r_v \in BV_{STUNE}$).

The function *select*$_I$ is equivalent to the one used in StratGEN (see Algorithm 6.2). Then, for any rule $r_r \in BR_{STUNE}, ar(r_r) = size$, therefore uniform recombination rules will be used over the whole population. On the other hand, the arity of a rule $r_v \in BV_{STUNE}$ is $ar(r_s) = 1$, that implies that behavioural variation rules will be applied to the last inserted member population, which corresponds to the offspring of the last $r_r$ rule selected.

#### 6.2.4.2 Insertion function

Once a new individual (i.e., parameter vector) is created, we have to insert it in the population. The StraTUNE *insert* function defines how to include a new individual on the current population by using a *fitness* function (see Section 5.4.5). This function is equivalent to the one used in StratGEN and shown in Algorithm 6.3.

The result of applying a rule $r_r \in BR_{STUNE}$ replaces the worst individual of the population. Then, variation rule $r_v \in BV_{STUNE}$ output replaces the worst element of the population, different from the last $r_r$ offspring inserted, if its fitness is better. Once again, the complete procedure is done each two iterations, i.e., it generates a new strategy using $r_r \in BR_{STUNE}$, to then its modifies it with the variation rule $r_v \in BV_{STUNE}$.

## 6.3 Experimental Results

After defining the background of both engines, we analyse the results of their best obtained strategies and the difference between both learning processes.

| Logic | Subset | Engine | (*Ltopi*[s], *Topi*[s]) (1,1) | | | (10,10) | | | (10,2400) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | solved | time[s] | total[s] | solved | time[s] | total[s] | solved | time[s] | total[s] |
| LIA | *known* | Z3 | 201 | 2.10 | 2.10 | 201 | 1.86 | 1.86 | **201** | **1.80** | **1.80** |
| | | **StratGEN** | **201** | **1.44** | **1.44** | **201** | **1.57** | **1.57** | 201 | 1.85 | 1.85 |
| | | StraTUNE | 201 | 1.84 | 1.84 | 201 | 1.83 | 1.83 | 201 | 1.94 | 1.94 |
| | *unknown* | Z3 | 180 | 5.89 | 14.89 | 182 | 8.09 | 78.09 | 185 | 2866.79 | 12466.79 |
| | | **StratGEN** | **189** | **4.62** | **4.62** | **189** | **5.09** | **5.09** | **189** | **6.00** | **6.00** |
| | | StraTUNE | 184 | 8.27 | 13.27 | 184 | 8.65 | 58.65 | 188 | 2864.14 | 5264.14 |
| LRA | *known* | Z3 | 331 | 6.69 | 14.69 | 333 | 10.83 | 70.83 | 337 | 491.00 | 5291.00 |
| | | **StratGEN** | **333** | **18.90** | **24.90** | **337** | **48.84** | **68.84** | **339** | **288.15** | **288.15** |
| | | StraTUNE | 328 | 6.11 | 17.11 | 335 | 21.14 | 61.14 | 337 | 644.90 | 5444.90 |
| | *unknown* | Z3 | 225 | 12.58 | 69.58 | 236 | 63.18 | 523.18 | 247 | 7324.12 | 91324.12 |
| | | StratGEN | **230** | **28.09** | **80.09** | **248** | **106.18** | **446.18** | 246 | 180.47 | 86580.47 |
| | | **StratGEN**$^{10}$ | 223 | 12.15 | 71.15 | 246 | 134.22 | 494.22 | **251** | **1798.06** | **76198.06** |
| | | StraTUNE | 228 | 21.58 | 75.58 | 245 | 76.72 | 446.72 | 251 | 5466.55 | 79866.55 |
| QF_LIA | *known* | **Z3** | 2879 | 637.01 | 3597.01 | 4102 | 5319.20 | 22689.20 | **5617** | **126486.39** | **659286.39** |
| | | StratGEN$^{10}$ | **2927** | **659.73** | **3571.73** | **4195** | **8564.35** | **25004.35** | 5510 | 125338.72 | 914938.72 |
| | | StraTUNE$^{10}$ | 2646 | 708.49 | 3901.49 | 4104 | 5844.00 | 23194.00 | 5508 | 134336.69 | 928736.69 |
| | *unknown* | Z3 | 81 | 37.82 | 258.82 | 110 | 200.39 | 2120.39 | 130 | 13230.82 | 426030.82 |
| | | StratGEN | 84 | 39.72 | 257.72 | 108 | 157.44 | 2097.44 | 127 | 10952.53 | 430952.53 |
| | | StratGEN$^{10}$ | 82 | 36.65 | 256.65 | 99 | 118.19 | 2148.19 | 131 | 29525.91 | 439925.91 |
| | | **StraTUNE** | **168** | **78.40** | **212.40** | **197** | **135.68** | **1185.68** | **208** | **18393.09** | **243993.09** |
| QF_LRA | *known* | Z3 | 1054 | 71.62 | 643.62 | 1173 | 494.45 | 5024.45 | 1530 | 116198.57 | 346598.57 |
| | | **StratGEN**$^{10}$ | **1110** | **98.59** | **614.59** | **1257** | **746.96** | **4436.96** | **1582** | **59430.71** | **165030.71** |
| | | StraTUNE$^{10}$ | 1029 | 71.39 | 668.39 | 1160 | 588.78 | 5248.78 | 1505 | 108981.88 | 399381.88 |
| | *unknown* | Z3 | 0 | 0.00 | 56.00 | 0 | 0.00 | 560.00 | 2 | 2886.58 | 132486.58 |
| | | **StratGEN** | **5** | **1.88** | **52.88** | **18** | **90.22** | **470.22** | **50** | **5793.71** | **20193.71** |
| | | StraTUNE | 0 | 0.00 | 56.00 | 0 | 0.00 | 560.00 | 2 | 1986.23 | 131586.23 |

Table 6.2 – SMT-LIB Benchmarks: Solving selected logic instances set using strategies generated by evolutionary programming engines.

### 6.3.1 Performance highlights

Table 6.2 summarises performance of Z3 solver over the selected logics by using its default strategy and the best strategies generated by StratGEN and StraTUNE. Let us remark the following:

1. In each instance subset, i.e., *known* and *unknown*, the engine which triggers Z3 best global result between all execution scenarios will be highlighted in **bold**.
2. Best result in each execution scenario will also be highlighted in **bold**.
3. An engine with a reduced learning set ($Lspct = 10\%$) to tackle a logic subset will be denoted with the number 10 as superscript of the engine, e.g., StraTUNE$^{10}$.

Note also that we mention engines performance referring to the effects of their best generated strategy in Z3 solver. The same principle applies to the engines that *outperform* Z3.

#### 6.3.1.1 Z3 improvements

Z3 performance is improved in most scenarios, 22 of 24 execution cases, either using StratGEN or StraTUNE generated strategies. In 1 of 8 logics sets, Z3 default configuration remains as top performer: `QF_LIA` *known*. For this logic subset, Z3 has a strong expert-designed strategy used to demonstrate the importance of *The Strategy Challenge in SMT* [10]. Therefore, outperforming Z3 in this case is not a trivial task.

Best engine is StratGEN, whose generated strategies have the best performance in 6 of 8 instances sets and 19 of 24 execution scenarios. With regards to StraTUNE, it outperforms all other engines in 1 of 8 logics: `QF_LIA` *unknown*. Despite of not being the best in the others scenarios, it outperforms Z3 default configuration in many other cases as: `LIA` *unknown* and the whole `LRA` instance set.

Major achievements include:

— StratGEN succeeds to solve entirely `LIA` *unknown* and `LRA` *known* subsets.

— StraTUNE dramatically outperforms Z3 in `QF_LIA` *unknown* logic subset solving 60% more instances and reducing the total solving time 43%.

— StratGEN dramatically improves Z3 performance in `QF_LRA` *unknown*, solving up to 86% of instances in set. Note that Z3 default configuration only solves 3% of instances. The time execution also is reduced 85%.

— StratGEN improves Z3 in `LRA` *unknown* and `QF_LRA` *known* logics. In the first logic, up to 11% of unsolved instances are addressed reducing around 18% of time execution. Meanwhile in the second case, up to 54% of unaddressed instances are solved in around *half the time* used by Z3 with its default configuration.

#### 6.3.1.2 Initial Strategy factor

A great difference between StratGEN and StraTUNE is their initial strategy ($Is$) and the role this element plays in each evolutionary process.

As $Is$ of StratGEN (see Figure 6.2) is inspired by `LIA`/`LRA` Z3 default strategy (see Figure 6.5a), it is thus similar, and we expected to perform well in those logics, as well as, in logics with a simpler strategies (e.g., `QF_LRA`). However, since StratGEN cannot change $Is$ structure to generate more complex strategies, then it fails to improve the hardest logic in the set, `QF_LIA`.

For StraTUNE, its $Is$ defines the size of the individuals, consequently the effectiveness of the engine. This is the main factor in the success of this engine in `QF_LIA` *unknown*, because Z3 default strategy includes an extensive explicitly defined parameter vector. On the other hand, it also explains the failure of the engine addressing the entire `QF_LRA` set, which has the simplest default strategy in Z3 (see Figure 6.5b).

Therefore, choosing an appropriate starting point must be carefully considered for the effectiveness of each engine.

#### 6.3.1.3 Structural rules vs Behavioural rules

Despite the engine dependence on its initial strategy, we observe that Structural Rules have a greater impact in the generation process than Behavioural Rules. This is given by better overall performance of StratGEN with respect to StraTUNE. Therefore, complex engines should be more focused on this type of rules, without leaving aside other alternatives.

Let us remark that we have defined a restricted set of both type of rules. While $R_{SGEN}$ does not include rules for modify the skeleton of its $Is$, $R_{STUNE}$ only includes rules to modify explicitly written parameters. Then, new rules should be included to take more advantage of the $Is$ information in order to generate more suitable strategies by changing their structure.

### 6.3.2 Learning process

Our engines rely on a learning procedure for generating efficient strategies. Thus, we explain how this phase supports obtained results.

#### 6.3.2.1 Learning Sample Size

As explained in Section 5.4.3, a reduced learning sample may be used by the engines, depending on the selected size of the set of instances or its complexity.

As shown in Table 6.2, the use of StraGEN[10] was able to improve default Z3 performance, and generate the best strategies for `LRA` *unknown* and `QF_LRA` *known* sets. Also, Z3 performance was improved in the two basic execution scenarios in the hardest set `QF_LIA` *known*.

Moreover, StratGEN$^{10}$ slightly outperforms its full learning sample counterpart (StratGEN) when both address the same logic subset. Regrettably, the use of a reduced learning sample does not trigger good performances in StraTUNE engine.

Thus, using a reduced learning sample helps tackling a set of problems whose evaluation is computationally expensive, or having an effective alternative for strategy generation process.

#### 6.3.2.2 Learning Variability

A relevant feature of the designed engines is their stochastic procedure to select modifying rules. Henceforth, engines were executed under different random seeds scenarios to check their behaviour. Let us remark that the best found strategies results were discussed in Section 6.3.1.

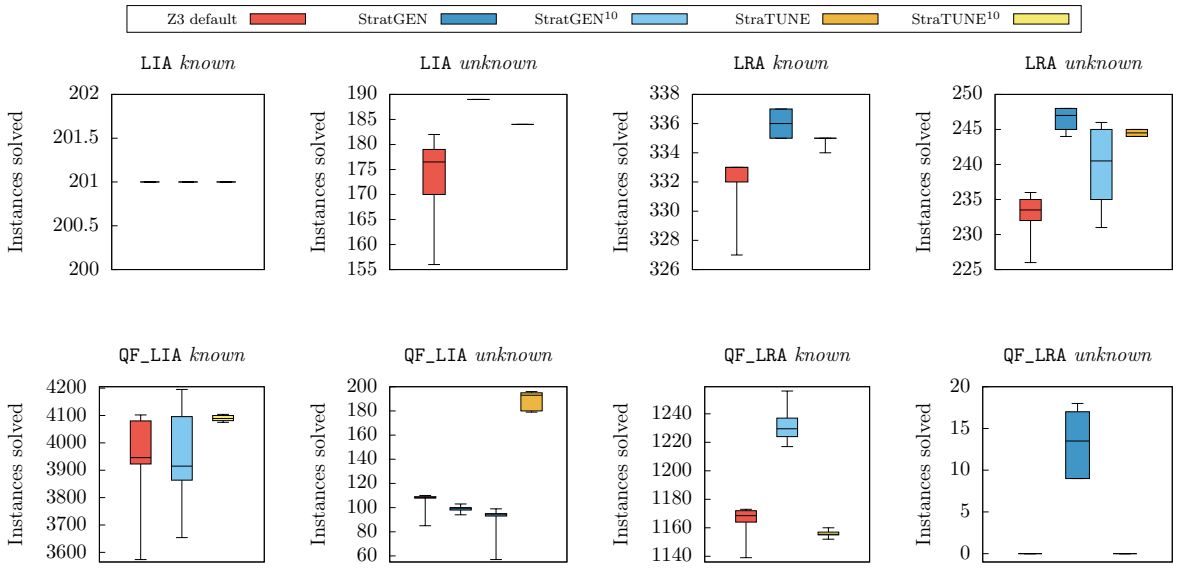



Figure 6.7 – Evolutionary Programming engine learning variability in the strategy generation process according to SMT logics with $Ltopi = 10$ seconds. Z3 values correspond to default execution with $Topi = Ltopi$.

Figure 6.7 shows performance variation of strategies generated by engines using different random seeds. We could observe, the performance of strategies depends on how modifying rules are selected. In this learning scenario ($Ltopi = 10$ seconds), the Z3 default configuration is outperformed in every logic. Also, StratGEN have better overall performance than StraTUNE. However, StraTUNE has less variability in all random scenarios, being more robust than StratGEN, whose performance is more affected by the selected random case, e.g., in `QF_LIA` logic.

To measure the results of the learning phase, we use a Two-Tailed Student T Test for means of paired samples with significance level of $\alpha = 0.05$. This test shows statistically significant

difference between mean performance of StratGEN and StraTUNE using ten executions with different random seed values. Note, our test uses the number of instances solved as main data. However, when both engines are completely tied in this metric, the analysis is done using time execution information.

| Logic | Subset | Type | Diff. Mean | RMSD | $t$ value | $p$ value | better | status |
|---|---|---|---|---|---|---|---|---|
| LIA | *known* | time | 0.43 | 0.02 | 27.26 | <**0.000001** | StratGEN | ✓ |
| | *unknown* | instances | -4.90 | 0.10 | 49.00 | <**0.000001** | StratGEN | ✓ |
| LRA | *known* | instances | -1.10 | 0.31 | -3.50 | **0.006745** | StratGEN | ✓ |
| | *unknown* | instances | -2.10 | 0.48 | -4.36 | **0.001829** | StratGEN | ✓ |
| QF_LIA | *known* | instances | 142.50 | 50.91 | 2.80 | **0.020746** | StraTUNE | ✓ |
| | *unknown* | instances | 90.90 | 2.34 | 38.84 | <**0.000001** | StraTUNE | ✓ |
| QF_LRA | *known* | instances | -76.60 | 4.060 | -18.90 | <**0.000001** | StratGEN | ✓ |
| | *unknown* | instances | -11.50 | 1.17 | -11.57 | <**0.000001** | StratGEN | ✓ |

Table 6.3 – Student T Test: Statistical significance between StratGEN and StraTUNE engines in the learning phase, with level $\alpha = 0.05$.

Table 6.3 summarises the results of the statistic test. StratGEN has a significantly better learning process in LIA, LRA, and QF_LRA sets. Meanwhile, StraTUNE has significantly better learning performance in QF_LIA logic. This results match the performance shown in Section 6.3.1.

## 6.4 Conclusions

In this chapter two evolutionary programming engines, driven by mutation-based rules, were introduced and implemented: StratGEN and StraTUNE. The former uses mostly structural variation rules, and the later behavioural variation modifiers.

We introduce two initial strategies for our engines, and shown how they affect their performance. We also discussed the importance of rules to generate more complex strategies, because both StratGEN and StraTUNE were not designed to modify the structure (i.e., skeletons) of their initial strategies.

Then, experimental results showed both engines succeeded to improve Z3 performance, but a higher impact were achieved when structural-oriented rules are applied. However, behavioural rules also help to generate efficient strategies.

Despite this, Z3 performance could not be improved in some scenarios, e.g., QF_LIA *known* subset. This is because, the generated strategies where not complex enough to address the instances in those sets.

# 7

# StratEVO: Evolving SMT Strategies

In the previous chapter, we have seen how rules define the success of an engine, specially in the case of fixed structure strategies. However, more efficient strategies can be obtained if their basic structure is modified, i.e., allowing them to evolve. This argument, in addition to the empirical guidelines inferred from the performance of designed evolutionary programming engines, serves as a starting point for the construction of a more sophisticated engine which allows to generate more complex strategies by evolving their structure.

## 7.1 StratEVO: A Tree-based Genetic Programming engine

Our third engine is a *Tree-based Genetic Programming* algorithm, called **StratEVO**. It aims at building more complex strategies by evolving most components in the initial strategy structure, i.e., by generating several different strategy structures. Thus, the evolutionary process is not restricted to terminal components as basic tactics or its parameters vectors. StratEVO is based on classic concepts of Genetic Programming [40, 41] and Grammar-based Genetic Programming [44, 45, 46], and relies on a complex set of modification, where Structural rules preponderate over Behavioural rules.

### 7.1.1 Initial Strategy, $Is$

StratEVO could start its evolutionary process from two different starting points:

1. A basic structure strategy inspired by recurrent skeleton and tactics present in Z3 default strategies. This strategy is used as $Is$ by the StratGEN engine.

2. The Z3 default strategy depending on the selected SMT logic. This initial strategy is used in StraTUNE engine.

We refer the reader to Section 6.1.1 and Section 6.2.1 respectively, where both initial strategy structures are explained in detail. Let us remark that StratEVO is able to change the structure of both initial strategies for generating new complex ones. It can also starts from a single-tactic strategy. This last process will be highly expensive because of the need of building a skeleton from scratch. Thus, we avoid it by using well-known strategies.

### 7.1.2 Rules

StratEVO relies on a set of rules, $R_{SEVO}$, composed of Structural and Behavioural rules, where the first type predominates. This is the first engine that includes genuine structural crossover over strategies, allowing to import and export sub-terms in order to generate different structure strategies. This operator is one of the core elements in the evolutionary process. Moreover, the amount of generic rules increase considerably, thus we group them according to their effects.

#### 7.1.2.1 Structural Variation Rules

Let $SV_{SEVO} \subset R_{SEVO}$ be the set of mutation-driven rules applied by StratEVO. This set covers several scenarios that evolutionary programming engines do not, including basic structure modifications.

##### 7.1.2.1.1 Variations on combinators

We use a set of four generic rules that may be applied in order to exchange strategy combinators or to delete them, changing thus the global structure of the strategy.

— Change combinator: two straightforward rules that change a combinator for another one of the same arity. Note that changing a `and-then` by a `or-else` changes the scope of a strategy, by interchanging a conjunction by a disjunction or viceversa. After the change, the strategy has a new scope, i.e., its semantics changed: it is able to address logic instances in a new way, different from the previous strategies. However, we must ensure that the strategy is still correct with regards to its own context. Therefore, resulting

strategies must satisfy *solver constraint* (`solver_c()`, see Section 5.2.2) requirements.

$$CC^1_{SEVO} : st[\texttt{and-then}_{/n}(st^n)] \rightarrow st[\texttt{or-else}_{/n}(st^n)] \; \{\texttt{solver\_c}(st[\texttt{or-else}_{/n}(st^n)])\}$$
$$CC^2_{SEVO} : st[\texttt{or-else}_{/n}(st^n)] \rightarrow st[\texttt{and-then}_{/n}(st^n)] \; \{\texttt{solver\_c}(st[\texttt{and-then}_{/n}(st^n)])\}$$

— Delete nested combinator: a generic rule that removes a substrategy root, i.e., a combinator, inside another combinator. It is clear that when both functions are `and-then` (resp. `or-else`), the nested combinator can be flattened without modifying the semantics of the strategy. However, when an `and-then` combinator is inside a `or-else` based composed tactic (or vice-versa) applying the rule changes the scope and the semantics of the strategy $st$.

$$DN_{SEVO} : \gamma_{/n_1+n_2+1}({st_1}^{n_1}, \gamma'_{/n}(st^n), {st_2}^{n_2}) \rightarrow \gamma_{/n_1+n_2+n}({st_1}^{n_1}, st^n, {st_2}^{n_2})$$

— Add strategy in combinator: this rule enables to extend a sequence of strategies that appears in a combinator. The semantics/scope of the strategy is thus changed with a `and-then` and may be changed with a `or-else`.

$$AS_{SEVO} : \gamma_{/n1+n2}({st_1}^{n_1}, {st_2}^{n_2}) \rightarrow \gamma_{/n1+n_2+1}({st_1}^{n_1}, st, {st_2}^{n_2})$$

#### 7.1.2.1.2 Variations on tactics

Several rules are used in order to exchange, modify, add, or delete tactics within the strategy structure. This means that heuristics, solvers, and probes can be modified with respect to syntactic constraints. Moreover, additional constraints must be checked, e.g., we must ensure that at least a solver is executed in the strategy (otherwise, the only possible result would be `unknown`), depending on the control structure induced by the combinators.

— Delete tactic in combinator: these three rules are similar and are used to either remove a solver (in this case, we have to verify that the new strategy is still a valid strategy), a heuristic, or a probe.

$$DT^s_{SEVO} : st[\gamma_{/n_1+n_2+1}({st_1}^{n_1}, s, {st_2}^{n_2})] \rightarrow st[\gamma_{/n_1+n_2}({st_1}^{n_1}, {st_2}^{n_2})] \; \{\texttt{solver\_c}(st[\gamma_{/n_1+n_2}({st_1}^{n_1}, {st_2}^{n_2})]) \wedge n_1 + n_2 \neq 0\}$$

$$DT^{h}_{SEVO} : \gamma_{/n_1+n_2+1}({st_1}^{n_1}, h, {st_2}^{n_2}) \rightarrow \gamma_{/n_1+n_2}({st_1}^{n_1}, {st_2}^{n_2}) \ \{n_1 + n_2 \neq 0\}$$
$$DT^{p}_{SEVO} : \gamma_{/n_1+n_2+1}({st_1}^{n_1}, p, {st_2}^{n_2}) \rightarrow \gamma_{/n_1+n_2}({st_1}^{n_1}, {st_2}^{n_2}) \ \{n_1 + n_2 \neq 0\}$$

— Modify tactic: basic generic rules, for changing basic tactics values (similar to StratGEN variation rules).

$$MT^{s}_{SEVO} : s \rightarrow s'$$
$$MT^{h}_{SEVO} : h \rightarrow h$$
$$MT^{p}_{SEVO} : p \rightarrow p$$

— Exchange tactics: generic rules which exchange a heuristic by a solver and viceversa. Note that, from a semantical point of view, only these changes are considered. Probes, due to their nature, are not compatible with such changes. If a heuristic is turned into a solver, the whole strategy satisfies *solver constraint* (`solver_c()`, see Section 5.2.2), thus there is no need to check this condition. On the other hand, if a solver turns into heuristic, it is strictly necessary to verify this property to avoid introducing ill-formed strategies into the evolutionary process.

$$ET^{hs}_{SEVO} : h \rightarrow s$$
$$ET^{sh}_{SEVO} : st[s] \rightarrow st[h]\{\texttt{solver\_c}(st[h])\}$$

#### 7.1.2.2 Structural Recombination Rules

Let $SR_{SEVO} \subset R_{SEVO}$ be the set of recombination exchange rules applied to sub-terms as it is usually done in Genetic Programming [40, 41]. This set is composed of the following generic rules:

— Term recombination: a sub-term of a strategy $st_1$, is exchanged with a sub-term of another strategy $st_2$.

$$TR_{SEVO} : (st_1[l_1], st_2[l_2]) \rightarrow (st_1[l_2], st_2[l_1]) \ \{\texttt{solver\_c}(st_1[l_2], st_2[l_1])\}$$

This generic rule generates an offspring of two new strategies, which generally have a different scope, i.e., semantic, with regards to his parents. Of course, offsprings must satisfy *solver constraint* requirements, and include all possible subterm exchanges between two strategies.

#### 7.1.2.3 Behavioural Variation (BV)

Let $BV_{SEVO} \in R_{SEVO}$ be a set of simple behavioural rules used by StratEVO. As StraTUNE (see Section 6.2), this set is driven by mutation based rules applicable over explicitly defined

(i.e., written) parameters in a strategy. Let us remark that this explicit value appears in the strategy within `using-params` or `try-for` modifier functions (set $\Delta$), and their values, e.g., tactic time limits are carefully handled as in evolutionary programming engines. Also remember, $EW$ is the set of parameters explicitly written in a strategy. This set is composed by two generic rules:

— Modify parameters: allows to change the value of the $k^{th}$ component of a tactic parameter vector, $\pi_k \in EW$, as seen in StraTUNE.

$$MP_{SEVO} : \delta(st, \pi) \rightarrow \delta(st, \pi')\{\texttt{compatible}(\pi, \pi') \wedge \pi_k \neq \pi'_k\}$$

Once again, note that this value modification implies $\pi'_k \in EW$.

— Restore default parameter vector: restores tactics configuration values to their default parameter vector by removing their explicitly written parameters.

$$RD_{SEVO} : \delta(st, \pi) \rightarrow st$$

Note that time limit parameter is handled in a different function (`try-for`) than other parameters (`using-params`). Thus, this generic rule must be applied twice to completely reset a tactic to its default configuration.

### 7.1.3 Solution Encoding

Since StratEVO needs the whole information of a strategy, we use first-order terms, encoded as trees, to represent them. Term grammar $\mathcal{G}_{Z3}$ can generate each valid strategy using its derivation tree (see Section 4.3.2). Thus, StratEVO can handle several type values existing in the tactics as well as in the parameters.

Figure 7.1 shows how strategies are represented as trees. The example strategy $S$ (Figure 7.1a) is represented as a tree (Figure 7.1b) where its components, following the corresponding color notation, portray the following:

— Basic tactics (in blue), where cross-circle ($\otimes$) are probes, circles ($\bigcirc$) are heuristics and squares ($\square$) are solvers.

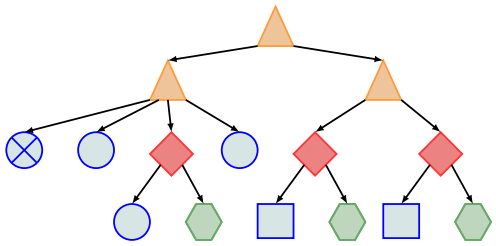
— Combinators functions (in orange) as triangles ($\triangle$). These symbols map to tactics composers, as `or-else` or `and-then`.

— Parameter modifier functions (in red) as diamonds ($\diamond$). This includes the parameter components (in green) as hexagons ($\hexagon$) modified by the function (`using-params` or `try-for`).

Note that this representation allows to apply easily all different StratEVO rules explained above, as shown in Example 7.1.

```
1  (and-then
2          (or-else
3                  (fail-if (not(is-ilp)))
4                  simplify
5                  (using-params
6                          ctx-simplify
7                   :max_depth 30
8                   :max_steps 5000000
9                  )
10                 split-clause
11         )
12         (and-then
13                 (try-for sat 100)
14                 (using-params
15                         smt
16                  :random-seed 100
17                 )
18         )
19 )
```

(a) User-defined strategy example.



(b) Strategy example encoded as simple tree.

Figure 7.1 – StratEVO Solution Encoding: Representing strategies as simple trees.

**Example 7.1** Let $S_1$ and $S_2$ be two identical strategies (shown in Figure 7.1), and the structural rules *Term Recombination* ($TR_{SEVO}$) and *Exchange Tactic* ($ET^{sh}_{SEVO}$).

The first rule can match any exchange of substrategies between $st_1$ and $st_2$. Thus, let $l_1$ be the heuristic `simplify-ctx` modified by $\texttt{using-params}_{/2}$ shown in Figure 7.1a (lines 5 to 9), and let $l_2$ be the `smt` solver modified by $\texttt{try\_for}_{/2}$ shown in the same figure (line 13). Then, the rule

$$TR_{SEVO} : (st_1[l_1], st_2[l_2]) \rightarrow (st_1[l_2], st_2[l_1])\ \{\texttt{solver\_c}(st_1[l_2], st_2[l_1])\}$$

is valid, and generates two well-formed offsprings with regards to the *solver constraint*. This crossover procedure is shown in Figure 7.2.

The second rule can match any exchange of a solver by a heuristic, while the resulting strategy is still well-formed and respects the *solver constraint*. Let $s$ be the solver in $st_1$ shown in Figure 7.1a (line 15). Then, the rule

$$ET^{sh}_{SEVO} : st[s] \rightarrow st[h]\{\texttt{solver\_c}(st[h])\} \qquad \diamondsuit$$

is valid $\forall\ h \in \texttt{Heuristic}$, generating valid strategies. This mutation procedure is shown in Figure 7.3.

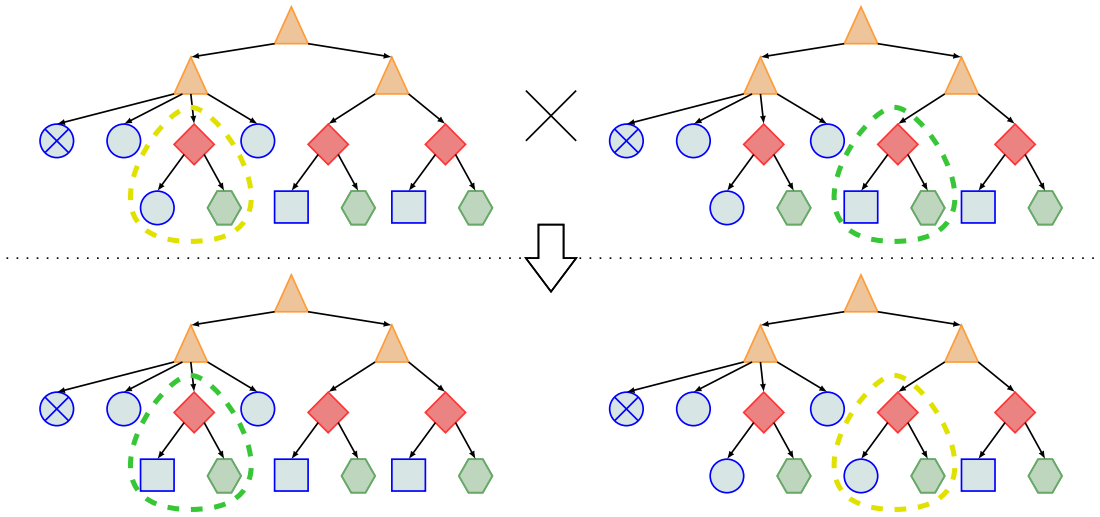


Figure 7.2 – Structural Recombination rule application, i.e., crossover, in a tree encoded strategy.

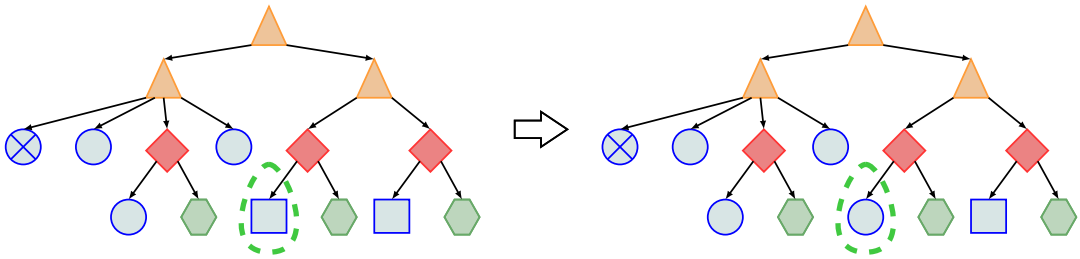


Figure 7.3 – Structural Variation rule application, i.e., mutation, in a tree encoded strategy.

### 7.1.4 Population

As the tree representation is defined for all strategies, we could use it in the whole evolutive process of StratEVO. Thus, an individual is a complete strategy represented as a tree. Then, the population is a set of strategies, i.e., a set of trees. The *population size* is defined as the double of the population size of StratGEN, i.e, $2 \times 10 = 20$ individuals.

The initial population is set up by applying indepently a set of randomly selected Structural Variation rules ($SV_{SEVO}$) over the initial strategy ($Is$). The amount of rules is equal to the size of the population, i.e., 20 rules. Then, population can evolve following the Evolutionary Algorithm scheme shown in Section 5.1.

### 7.1.5 Selection functions

Selection functions are in charge of choosing, at some point of the evolution process (i.e., iteration), a rule from $R_{SEVO}$ and a set of individuals to apply it, in order to generate a new strategy which could differ dramatically from the structure of its parents or from the initial population strategies.

**Algorithm 7.1:** StratEVO $select_R$ function

**Input:** a set of rules $R_{SEVO}$
**Output:** A rule set $r$

```
1: if iteration number odd then
2:     r ← random(SR_SEVO)
3: else
4:     r ← random(SV_SEVO ∪ BV_SEVO, 2)
5: end if
6: return r
```

Algorithm 7.1 shows how the $select_R$ function picks rules from $R_{SEVO}$. First, it selects a structural recombination rule ($r_r \in SR_{SEVO}$). Then, in the following generation, it chooses two variation rules ($r_v \in SV_{SEVO} \cup BV_{SEVO}$).

Meanwhile, the $select_I$ function (shown in Algorithm 7.2) chooses a set of individuals to apply the selected rules. If the rule arity is two, individuals are selected by means of fitness using a classic tournament selection operator. Here, each individuals is selected from a semifinal branch composed of four random individual members. Of course, individuals can only participate in one branch. Then, if the rule arity is one, the last inserted member of the population and its sibling are selected.

Note that the arity of $r_r \in SR_{SEVO}$ is always $ar(r_r) = 2$. Then, tournament selection is exclusively performed to choose the parents for crossover operation. Moreover, $r_v \in SV_{SEVO} \cup$

**Algorithm 7.2:** StratEVO $select_I$ function

**Input:** an amount of individuals, $n$
a population, *population*
a fitness function *fitness*
**Output:** a set of individual $ind^n$

```
1: if n = 2 then
2:    ind^n ← tournament_selection(population,fitness,2)
3: else
4:    last ← last_inserted(population)
5:    ind^n ← last, sibling(last)
6: end if
7: return ind^n
```

$BV_{SEVO}$ arity is $ar(r_v) = 1$, therefore the variation rules will be exclusively applied to the offspring of a recombination.

### 7.1.6 Insert function

Once a new strategy is generated, we need to insert it into the current population. The StratEVO *insert* function integrates new strategies by means of fitness.

**Algorithm 7.3:** StratEVO *insert* function

**input:** an individual set, $Ind'^{n'}$
a population, *population*
a fitness function *fitness*
**output:** a new *population* generation

```
1: new ← best(Ind'^{n'})
2: if iteration number odd then
3:    worst ← worst(population,fitness)
4:    population ← replace(worst,new)
5: else
6:    last ← last_inserted(population)
7:    worst ← worst(population-last,fitness)
8:    population ← replace(worst,new,fitness)
9: end if
10: return population
```

Algorithm 7.3 shows how the *insert* function proceeds. If new strategies were generated by means of recombination rules, the best offspring replaces the worst population member. But, if those strategies are generated using mutation-driven rules, the best one replaces the worst individual (different from the last inserted) if fitness is better. Note that a complete sequence of crossover and mutation is done every two generations.

## 7.2 Experimental results

After defining the background of StratEVO, we analyse the results of the best strategies obtained by the engine using two different initial strategies. Then, the difference of the learning processes between both version is discussed.

### 7.2.1 Performance highlights

As shown in Table 7.1, we analyse the performance of Z3 solver over the selected logics using its default strategy and the best strategies generated by StratEVO. We also include the best performer engine so far from previous chapters. Let us remark the following:

1. In each logic subset, i.e., *known* and *unknown*, the engine which triggers Z3 best global

| Logic | Subset | Engine | ($Ltopi$[s], $Topi$[s]) (1,1) | | | (10,10) | | | (10,2400) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | solved | time[s] | total[s] | solved | time[s] | total[s] | solved | time[s] | total[s] |
| LIA | *known* | Z3 | 201 | 2.10 | 2.10 | 201 | 1.86 | 1.86 | 201 | 1.80 | 1.80 |
| | | *StratGEN* | 201 | 1.44 | 1.44 | 201 | 1.57 | 1.57 | 201 | 1.85 | 1.85 |
| | | StratEVO$_{FS}$ | 201 | 1.43 | 1.43 | 201 | 1.39 | 1.39 | 201 | 1.40 | 1.40 |
| | | **StratEVO$_{Z3}$** | **201** | **1.38** | **1.38** | **201** | **1.37** | **1.37** | **201** | **1.37** | **1.37** |
| | *unknown* | Z3 | 180 | 5.89 | 14.89 | 182 | 8.09 | 78.09 | 185 | 2866.79 | 12466.79 |
| | | *StratGEN* | 189 | 4.62 | 4.62 | 189 | 5.09 | 5.09 | 189 | 6.00 | 6.00 |
| | | StratEVO$_{FS}$ | 189 | 4.90 | 4.90 | 189 | 4.85 | 4.85 | 189 | 4.83 | 4.83 |
| | | **StratEVO$_{Z3}$** | **189** | **4.51** | **4.51** | **189** | **4.47** | **4.47** | **189** | **4.47** | **4.47** |
| LRA | *known* | Z3 | 331 | 6.69 | 14.69 | 333 | 10.83 | 70.83 | 337 | 491.00 | 5291.00 |
| | | *StratGEN* | 333 | 18.90 | 24.90 | 337 | 48.84 | 68.84 | 339 | 288.15 | 288.15 |
| | | StratEVO$_{FS}$ | 333 | 7.24 | 13.24 | 337 | 26.98 | 46.98 | 339 | 199.26 | 199.26 |
| | | **StratEVO$_{Z3}$** | **333** | **6.96** | **12.96** | **337** | **23.94** | **43.94** | **339** | **193.25** | **193.25** |
| | *unknown* | Z3 | 225 | 12.58 | 69.58 | 236 | 63.18 | 523.18 | 247 | 7324.12 | 91324.12 |
| | | *StratGEN*$^{10}$ | 223 | 12.15 | 71.15 | 246 | 134.22 | 494.22 | 251 | 1798.06 | 76198.06 |
| | | **StratEVO$_{FS}$** | 231 | 16.50 | 67.50 | 250 | 126.56 | 446.56 | **265** | **7635.47** | **48435.47** |
| | | StratEVO$^{10}_{FS}$ | 211 | 7.99 | 78.99 | 216 | 43.52 | 703.52 | 219 | 138.54 | 151338.54 |
| | | StratEVO$_{Z3}$ | **235** | **17.62** | **64.62** | **251** | **207.56** | **517.56** | 265 | 8621.53 | 49421.53 |
| | | StratEVO$^{10}_{Z3}$ | 228 | 14.18 | 68.18 | 248 | 123.41 | 463.41 | 262 | 6243.20 | 54243.20 |
| QF_LIA | *known* | ***Z3*** | 2879 | 637.01 | 3597.01 | 4102 | 5319.20 | 22689.20 | **5617** | **126486.39** | **659286.39** |
| | | StratEVO$^{10}_{FS}$ | 2913 | 646.29 | 3572.29 | 3975 | 4725.65 | 23365.65 | 5612 | 158508.51 | 703308.51 |
| | | StratEVO$^{10}_{Z3}$ | **2975** | **674.71** | **3538.71** | **4220** | **6767.21** | **22957.21** | 5536 | 104462.86 | 831662.86 |
| | *unknown* | Z3 | 81 | 37.82 | 258.82 | 110 | 200.39 | 2120.39 | 130 | 13230.82 | 426030.82 |
| | | *StraTUNE* | 168 | 78.40 | 212.40 | 197 | 135.68 | 1185.68 | 208 | 18393.09 | 243993.09 |
| | | StratEVO$_{FS}$ | 82 | 38.82 | 258.82 | 109 | 193.58 | 2123.58 | 128 | 16597.24 | 434197.24 |
| | | StratEVO$^{10}_{FS}$ | 82 | 37.83 | 257.83 | 137 | 252.94 | 1902.94 | 169 | 9303.60 | 328503.60 |
| | | **StratEVO$_{Z3}$** | 197 | 33.25 | 138.25 | **198** | **41.21** | **1081.21** | **210** | **13617.78** | **234417.78** |
| | | StratEVO$^{10}_{Z3}$ | **197** | **24.99** | **129.99** | 197 | 339.17 | 1389.17 | 207 | 11974.85 | 239974.85 |
| QF_LRA | *known* | Z3 | 1054 | 71.62 | 643.62 | 1173 | 494.45 | 5024.45 | 1530 | 116198.57 | 346598.57 |
| | | *StratGEN*$^{10}$ | **1110** | **98.59** | **614.59** | 1257 | 746.96 | 4436.96 | 1582 | 59430.71 | 165030.71 |
| | | **StratEVO$^{10}_{FS}$** | 1108 | 98.93 | 616.93 | **1288** | **881.52** | **4261.52** | **1583** | **44576.55** | **147845.73** |
| | | StratEVO$^{10}_{Z3}$ | 1110 | 100.15 | 616.15 | 1255 | 762.34 | 4472.34 | 1578 | 54312.23 | 169512.23 |
| | *unknown* | Z3 | 0 | 0.00 | 56.00 | 0 | 0.00 | 560.00 | 2 | 2886.58 | 132486.58 |
| | | ***StratGEN*** | 5 | 1.88 | 52.88 | 18 | 90.22 | 470.22 | **50** | **5793.71** | **20193.71** |
| | | StratEVO$_{FS}$ | **16** | **15.84** | **55.84** | **38** | **140.29** | **320.29** | 38 | 1890.02 | 45090.02 |
| | | StratEVO$_{Z3}$ | 0 | 0.00 | 56.00 | 1 | 4.14 | 554.14 | 34 | 20165.82 | 72965.82 |

Table 7.1 – SMT-LIB Benchmarks: Solving selected logic instances set using strategies generated by StratEVO.

result between all execution scenarios will be highlighted in **bold**.

2. The best engine from previous chapter will be highlighted in *italic*.
3. The best result in each execution scenario will also be highlighted in **bold**.
4. As StratEVO has two choices for the initial strategy, we use Z3 to denote the $Is$ as the default Z3 strategy for a given logic, or FS if the StratGEN fixed skeleton initial strategy is chosen. This will be noted as subscript of the engine, e.g. StratEVO$_{FS}$.
5. If an engine with a reduced learning set ($Lspct = 10\%$) for a logic subset, it will be denoted with the number 10 as superscript of the engine, e.g., StratEVO$^{10}_{Z3}$.
6. Despite the addition a set of behavioural variation rules, we still consider StratEVO as a structural-based algorithm. Therefore, we use structural rules execution scenario (see Section 5.4.3).

Note also that we mention engines performance referring to the effects of their best generated strategy in Z3 solver. The same principle applies to the engines that *outperforms* Z3.

#### 7.2.1.1 Z3 improvements

StratEVO turns to be the best engine in 6 of 8 scenarios, and it also improves Z3 performance in 22 of 24 selected execution cases, outperforming most of StratGEN and StraTUNE improvements. The two logics for which StratEVO could not improve the current best results are `QF_LIA` *known* and `QF_LRA` *unknown* logic subsets. The former is a complex case to address, which demonstrates the huge knowledge inverted by Z3 developers to show the importance of *The Strategy Challenge in SMT* [10]. The later represents the best output generated by the StratGEN engine, which StratEVO could not match despite of the amount of cases covered by its rules.

Major achievements include:

— A considerable time reduction in sets completely solved. In `LIA` set time consumption is reduced by up to 25% with regards to StratGEN. Meanwhile, in `LRA` *known* this amount reaches 33%.

— `LRA` *unknown* best performer is StratEVO. StratEVO considerably outperforms StratGEN, solving 45% of their unaddressed instances. This represents 4% of the total instances in the set. The execution time is also reduced, by around 36%.

— It obtains first competitive result in hardest logic set, `QF_LIA` *known*. StratEVO is slightly worse than the best performer in the set, solving only 1% less instances.

— It slightly outperforms best engine in `QF_LIA` *unknown* and `QF_LRA` *known* sets, solving up to 2% of unresolved instances, and reducing execution time up to 11%.

#### 7.2.1.2 Initial Strategy factor

Initial strategy also plays a great role in StratEVO performance, but not in every scenario as in evolutionary programming engines, i.e., StratGEN and StraTUNE.

Both initial strategies allows to reach good configurations, and this helps Z3 to solve the same amount of strategies in `LIA` and `LRA` sets. There exist a slight difference in time consumption (between 2% and 7%) using Z3 default strategies as starting point instead of the fixed skeleton strategy.

These behaviour changes in the remaining benchmarks set. In `QF_LIA`, the use of Z3 default strategies allows StratEVO to generate more efficient strategies by leading in 5 of 6 execution cases, and being dramatically different in the *unknown* set. This is because Z3 default strategy contains much more information than the StratGEN fixed skeleton strategy to support the engine. But in `QF_LRA`, the opposite happens. In this case, the Z3 default strategy lacks of knowledge to be exploited, and therefore its evolutive procedure is more expensive and weaker.

Thus, the initial strategy is still an important issue to address in order to generate more sophisticated strategies.

### 7.2.2 Learning process

As StratEVO relies on a learning procedure to generate efficient and complex strategies, we explain how this phase supports obtained results.

#### 7.2.2.1 Learning sample size

The reduced learning sample proves to be a great alternative for StratEVO, especially in cases when starting strategies provide or contain relevant information for the evolutive process. For example, in `QF_LIA` *unknown* set, StratEVO$^{10}_{Z3}$ outperforms the best evolutionary engine performance, and is slightly worse than StratEVO$_{Z3}$. With regards to StratEVO$_{FS}$, the use of reduced learning sets help dramatically to improve performance. Note that StratEVO$_{FS}$ could not even outperform Z3 default strategy, meanwhile StratEVO$^{10}_{FS}$ solves 41 more instances, representing an improvement of 32%.

In scenarios where only reduced size sample is used, i.e., `QF_LIA` *known* and `QF_LRA` *known*, StratEVO outperforms the best results previously founded in Chapter 6.

Thus, using a reduced learning sample still helps addressing scenarios whose evaluation is computationally expensive, or having an effective alternative for the strategy generation process.

#### 7.2.2.2 Learning variability

StratEVO is a stochastic procedure in which rules are selected randomly. Thus, the engine learning phase was executed several times with different random seeds values.

Figure 7.4 shows the variability of the learning phase using StratEVO. We observe that in most cases, the use of Z3 default strategies implies a more robust engine procedure with regards to the stochastic engine component, i.e., random seed, and an overall better performance than Z3 default configuration. On the other hand, the use of the StratGEN fixed skeleton strategy is considerably unstable as shown in the whole `LRA` logic and `QF_LIA` *known* benchmark sets, and is outperformed in most scenarios.

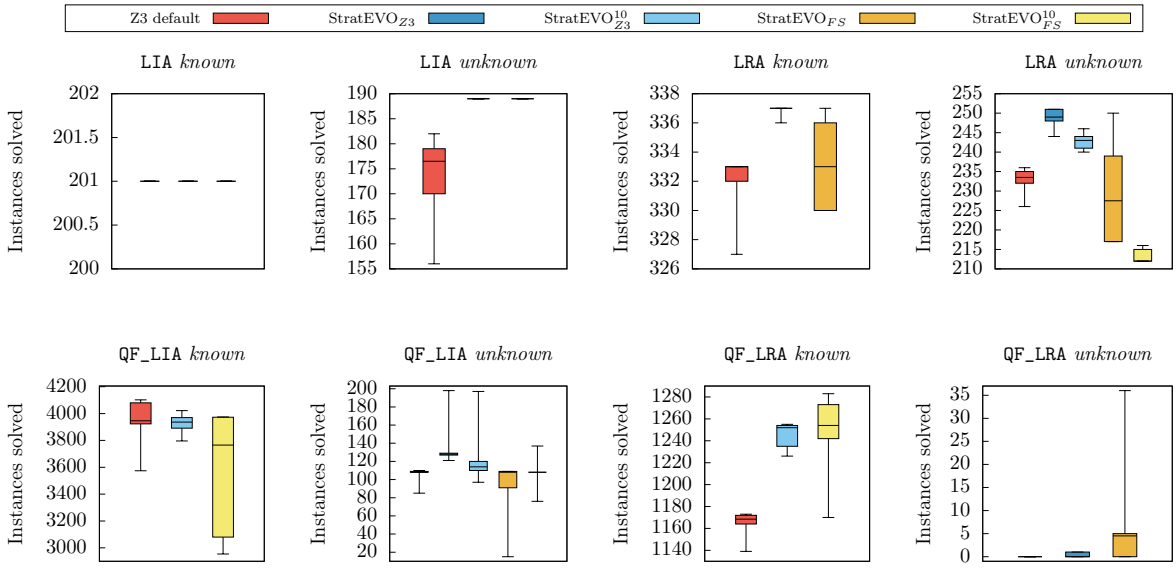



Figure 7.4 – StratEVO learning variability in the strategy generation process according to SMT logics with $Ltopi$ of 10 seconds. Z3 values correspond to default execution with $Topi = Ltopi$.

We use a Two-Tailed Student T Test for means of paired samples, with significance level of $\alpha = 0.05$, to measure the average performance of the two StratEVO versions: StratEVO$_{Z3}$ and StratEVO$_{FS}$. This test stands statistically significant difference between the performance of both alternatives, being executed ten times with different random seed values. Note that our test mainly uses instance solving data. However, when both engines are completely tied in this metric, the analysis is done using time execution information.

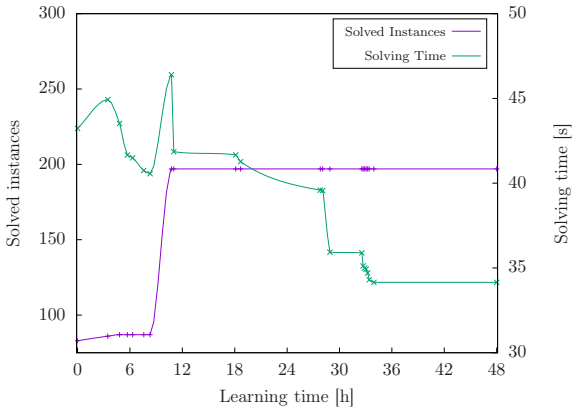
Table 7.2 summarises the results of the statistic test, where StratEVO$_{Z3}$ has a significantly better learning phase in the cases it wins with exception for `LIA` *known* where no difference could be statistically stated. Meanwhile, StratEVO$_{FS}$ cannot be stated as significantly better than its counterparts in `QF_LRA` logic. This analysis matches with the performance shown in Section 7.2.1.

| Logic | Subset | Type | Diff. Mean | RMSD | $t$ value | $p$ value | better | status |
|---|---|---|---|---|---|---|---|---|
| LIA | *known* | time | 0.17 | 0.56 | 2.20 | **0.055497** | StratEVO$_{Z3}$ | ✗ |
| | *unknown* | time | 0.96 | 10.76 | 2.77 | **0.021747** | StratEVO$_{Z3}$ | ✓ |
| LRA | *known* | instances | -3.7 | 98.10 | -3.54 | **0.006275** | StratEVO$_{Z3}$ | ✓ |
| | *unknown* | instances | -19.6 | 1418.40 | -4.94 | **0.000805** | StratEVO$_{Z3}$ | ✓ |
| QF_LIA | *known* | instances | -310.3 | 1682208.10 | -2.27 | **0.049390** | StratEVO$^{10}_{Z3}$ | ✓ |
| | *unknown* | instances | -36.9 | 15784.90 | -2.79 | **0.021182** | StratEVO$_{Z3}$ | ✓ |
| QF_LRA | *known* | instances | 5.6 | 6210.40 | 0.67 | **0.517162** | StratEVO$^{10}_{FS}$ | ✗ |
| | *unknown* | instances | 6.2 | 1039.60 | 1.82 | **0.101420** | StratEVO$_{FS}$ | ✗ |

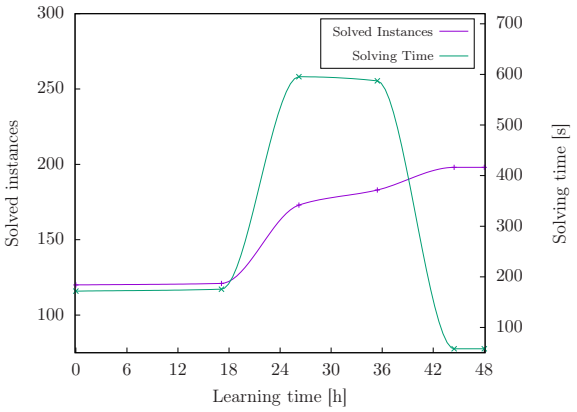
Table 7.2 – Student T Test: Statistical significance between initial strategies of StratEVO in the learning phase, with level $\alpha = 0.05$.

#### 7.2.2.3 Learning progress

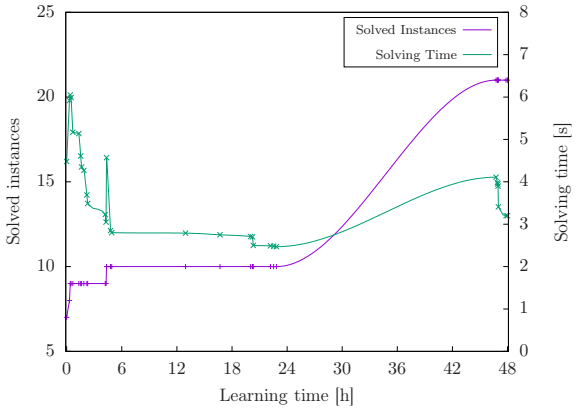
Engines have a parameter that defines a maximum learning time, called *Learning time budget* (*Lbt*), which is arbitrary set in 2 days. We analyse StratEVO behaviour in order to check if this limit restricts the engine potential or if it is overestimated.



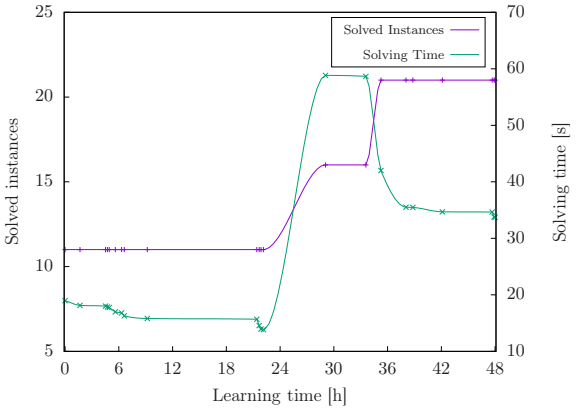

(a) StratEVO$_{Z3}$ with $Ltopi = 1$[s]




(b) StratEVO$_{Z3}$ with $Ltopi = 10$[s]




(c) StratEVO$^{10}_{Z3}$ with $Ltopi = 1$[s]




(d) StratEVO$^{10}_{Z3}$ with $Ltopi = 10$[s]

Figure 7.5 – StratEVO learning progress in QF_LIA *unknown.*

Figure 7.5 shows StratEVO learning progress in `QF_LIA` *unknown* logic subset, one of the most improved benchmarks set with regards to Z3 default performance. Here, we could observe improvements occurring during most the learning procedure, specially between the twelfth and the fortieth second hour of learning. Thus, the $Ltb$ bound allows to obtain most relevant improvements of engines which usually occurs during the second day of the learning phase.

## 7.3 Conclusions

In this chapter we presented StratEVO, a Tree-based Genetic Programming engine which generates complex SMT strategies for the Z3 solver.

It relies on a big set of rules which covers several ways for modifying strategies structures, based on classic Genetic Programming operators, i.e., mutation and crossover.

By repairing Z3 default and StratGEN initial strategies, StratEVO is able to outperforms the current best engines in most scenario. Regrettably, Z3 default configuration is still the best available for `QF_LIA` *known* set. However, a first competitive result is obtained by means of StratEVO.

Learning configuration was also discussed, proving that a reduced learning set is a viable alternative to generate efficient strategies. Most of the strategies improvement occurs between the twelfth and the fortieth second hour of the learning process, thus also validating two days learning budget time.

# 8

# Revisiting StratEVO: Cooperative Schemes for Evolution

In this chapter, we analyse some schemes for cooperation between behavioural rules and structural rules in order to add them in StratEVO. In the previous chapters, the engines were driven by one type of rules, and when both types were available one of them was randomly selected.

## 8.1 Rules

Modifying rules are the main component defining our cooperative schemes. Thus, we define the set of rules to be applied. Let $R_{COOP}$ be the set of rules used for cooperative engines. Rules are classified according to some types presented previously in Chapter 5.

— Structural Variation Rules: let $SV_{COOP} \subset R_{COOP}$ be a set of mutation-driven rules used to change strategy structure components. This set covers modification scenarios used by StratEVO and, consequently, StratGEN. We refer the reader to Section 7.1.2.1 for detailed rule explanations.

— Structural Recombination Rules: let $SR_{COOP} \subset R_{COOP}$ be the set of recombination exchange rules applied between sub-terms as it is usually done in Genetic Programming [40, 41]. This set is composed by the rule *Term Recombination* presented in StratEVO. We refer the reader to Section 7.1.2.2 for a detailed formalisation.

— Behavioural Variation Rules: let $BV_{COOP} \in R_{COOP}$ be a set of simple behavioural rules that allow to modify tactics parameter vectors. This set is composed by the same rules defined for StratEVO in Chapter 7. Thus, we refer the reader to Section 7.1.2.3 for a complete explanation.

— Behavioural Recombination Rules: let $BR_{COOP} \subset R_{COOP}$ be a set of generic Behavioural Recombination rules which generates a new strategy parameter vector. This set is composed by the same rule (i.e., *Uniform Recombination*) defined for StraTUNE. Thus, we refer the reader to Section 6.2.2.2 for a detailed explanation. Let us remark that $UR_{StraTUNE}$ can be interpreted as a union of mutation operators in each parameter, i.e., behavioural variation rules. Then, $UR_{StraTUNE}$ can be applied to a single individual in a population of strategies with different structures by using the domain of the explicitly written parameters of the strategy.

## 8.2 Cooperative Schemes

In order to integrate Structural and Behavioural rules in a hybrid environment, we define several cooperative schemes.

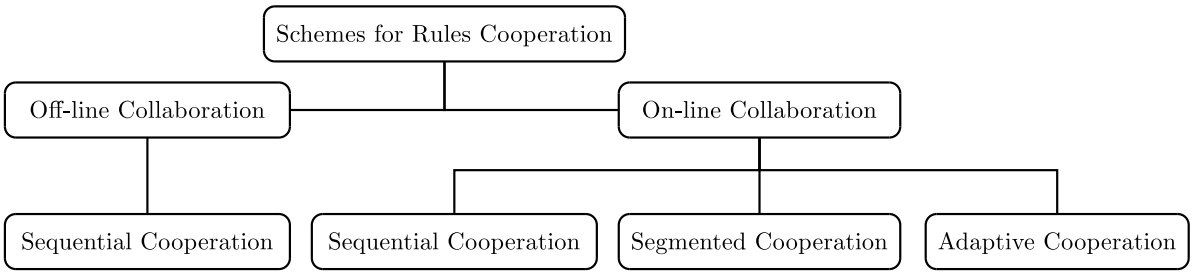



Figure 8.1 – Cooperative Schemes for Modifying Rules.

As Figure 8.1 shows, we can classify these schemes in two categories: Off-line Collaboration and On-line Collaboration.

### 8.2.1 Off-line Collaboration

We call *off-line collaboration* the process of using two different sets of rules in different environments, i.e., engines. This category has only one available scheme: Sequential Cooperation. By means of this work, the outputs of our different engines, must be chained. Therefore, classic schemes as Parallel Collaboration are discarded.

#### 8.2.1.1 Sequential Cooperation

Our first collaborative scheme aims at integrating sequentially the process of two engines, managing different types of rules.

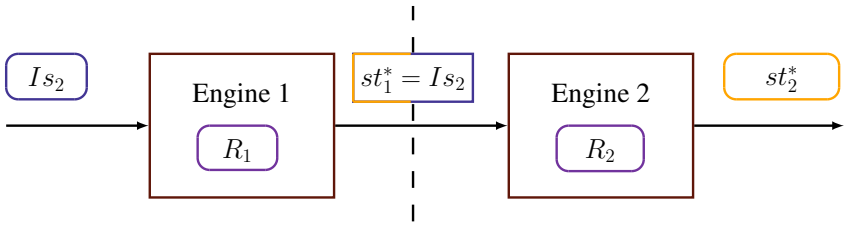



Figure 8.2 – Off-line Sequential Cooperative Scheme.

Figure 8.2 shows the procedure of this off-line sequentialisation. A predefined engine generates an optimised strategy $st_1^*$ with regards to a set of rules $R_1 \subset R_{COOP}$. Then, a second engine uses $st_1^*$ as an initial strategy to generate an improved strategy $st_2^*$ by means of a rule set $R_2 \in R_{COOP}$. Note that $R_1 \cup R_2 = R_{COOP}$ and $R_1 \cap R_2 = \varnothing$, and thus for our scope we have two options:

1. The first engine uses rules from $R1 = SV_{COOP} \cup SR_{COOP}$, consequently the second engine uses rules from $R_2 = BV_{COOP} \cup BR_{COOP}$.
2. The first engine uses rules from $R1 = BV_{COOP} \cup BR_{COOP}$, consequently the second engine uses rules from $R_2 = SV_{COOP} \cup SR_{COOP}$.

We discard the second scenario, because if behavioural rules are applied first their effects can be considerably reduced by changing the strategy structure, and thus the tactics semantics. This problem does not occur in the first sequential case, where an optimised structure is calibrated without risking the change of its structure.

### 8.2.2 On-line Collaboration

We call *on-line collaboration* the process of using two different sets of rules in the same environment, i.e., engine. This category has three available schemes: Sequential, Segmented and Adaptive Cooperation.

#### 8.2.2.1 Sequential Cooperation

In this scheme, the sequentialisation of rules occurs inside an engine. Thus, the type of rules are equally distributed between both sets.

Figure 8.3 shows the procedure of this on-line sequentialisation. A predefined engine generates an optimised strategy $st^*$ by applying sequentially rules from $R_1$ and $R_2$ ($R_1, R_2 \subset R_{COOP}$) starting from a initial strategy $Is$. Indeed, this sequentialisation is applied several times

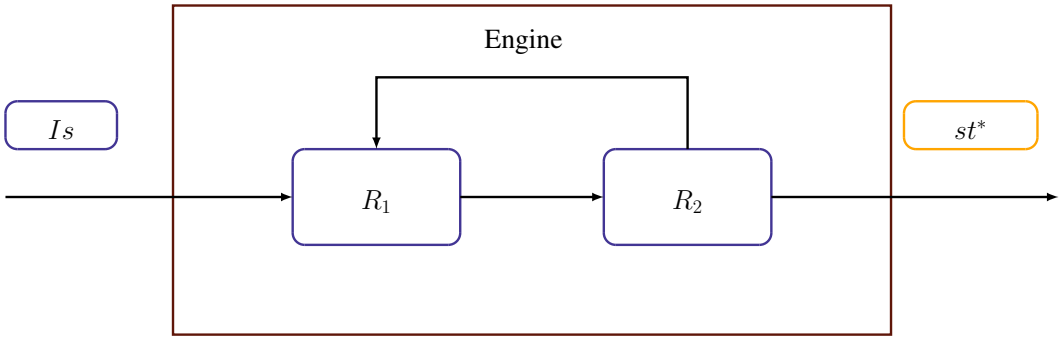



Figure 8.3 – On-line Sequential Cooperative Scheme.

inside the engine because of their iterative nature. Once again, let us remark that $R_1 \cup R_2 = R_{COOP}$ and $R_1 \cap R_2 = \varnothing$. Therefore, as well as in their off-line counterpart, there are two options (see Section 8.2.1.1), and we select the following case:

— First we apply rules from $R1 = SV_{COOP} \cup SR_{COOP}$, consequently we apply rules from $R_2 = BV_{COOP} \cup BR_{COOP}$.

Let us remark, if behavioural rules are applied first, their effects can be considerably reduced by changing strategy structure, thus tactics semantics.

#### 8.2.2.2 Segmented Cooperation

The second scheme, uses a segmented sequence of rules. Thus, one kind of rules is applied during a certain number of times, then a second type of rules is used for another amount of times.

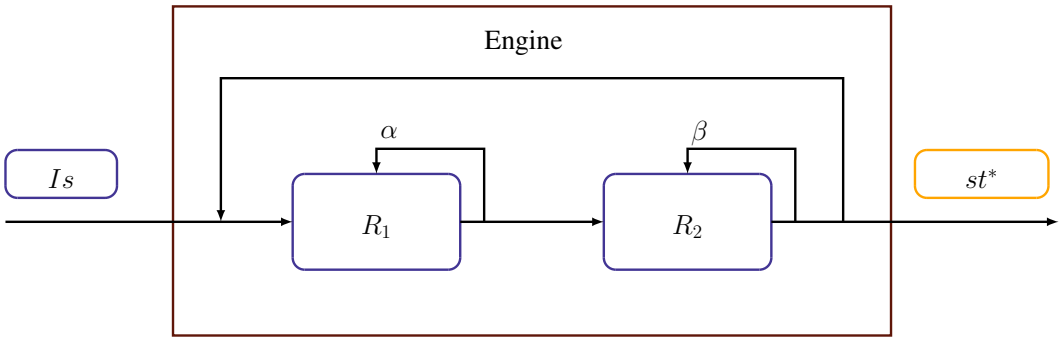



Figure 8.4 – On-line Segmented Cooperative Scheme.

The segmented sequentialisation procedure is shown in Figure 8.4. An engine generates an optimised strategy $st^*$ by applying sequentially rules from $R_1$ and $R_2$ ($R_1, R_2 \subset R_{COOP}$) starting for an initial strategy $Is$. The first stage is to apply $\alpha$ times rules from $R_1$, and then apply $\beta$ times rules from $R_2$ in $\beta$ times. In both stages, rules are randomly selected. Of course, this sequentialisation is applied several times inside the engine because of their iterative nature.

Once again, let us remark that $R_1 \cup R_2 = R_{COOP}$ and $R_1 \cap R_2 = \varnothing$. Therefore, as well as with past schemes, we are still working with the same sequentialisation:

— First we apply rules from $R1 = SV_{COOP} \cup SR_{COOP}$, then we apply rules from $R_2 = BV_{COOP} \cup BR_{COOP}$.

In this thesis, $\alpha$ and $\beta$ values are configured to give 60% priority to Structural Modification rules. Therefore, we set $\alpha = 3$ and $\beta = 2$.

#### 8.2.2.3 Adaptive Cooperation

Our last online cooperative scheme is an adaptive procedure. In this, given a probability, a certain type of rule is applied. Then, depending on their effects, the chance of selecting another rule of the same type, increases or decreases.

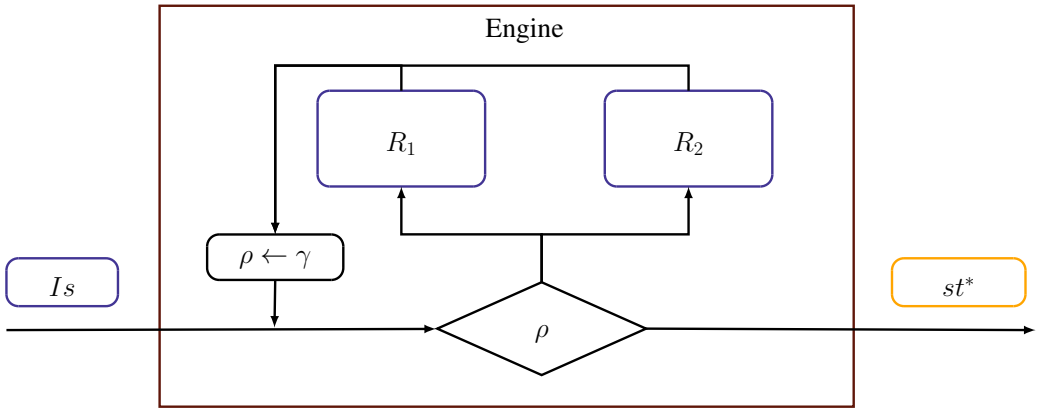



Figure 8.5 – On-line Adaptive Cooperative Scheme.

As shown in Figure 8.5, an engine generates an optimised strategy $st^*$ by applying sequentially rules from $R_1$ and $R_2$ ($R_1, R_2 \subset R_{COOP}$) starting from an initial strategy $Is$. Given a probability $\rho$, a rule from $R_1$ is selected. Otherwise, a rule from $R_2$ is chosen. Depending on the rule impact in the strategy generation process, the value of $\rho$ variates as follows:

— If a rule from $R_1$ triggers an improvement of the best solution, $\rho$ value increases of $\gamma$; otherwise it decreases of $\gamma$.

— If a rule from $R_2$ triggers an improvement of the best solution, $\rho$ value decreases of $\gamma$; otherwise it increases of $\gamma$.

Once again, let us remark that $R_1 \cup R_2 = R_{COOP}$ and $R_1 \cap R_2 = \varnothing$. In this thesis, $\rho$ value is configured to give 60% priority to Structural Modification rules at the beginning, while increase/decrease variation corresponds to 5% of chance change. Therefore, we set $\rho = 0.6$ and $\gamma = 0.05$.

## 8.3 SequencEVO: Off-line Sequential Cooperation Engine

Our first engine called, *SequencEVO*, use off-line sequential cooperation to evolve. It combines previously designed engines StratEVO (see Chapter 7) and StraTUNE (see Section 6.2).

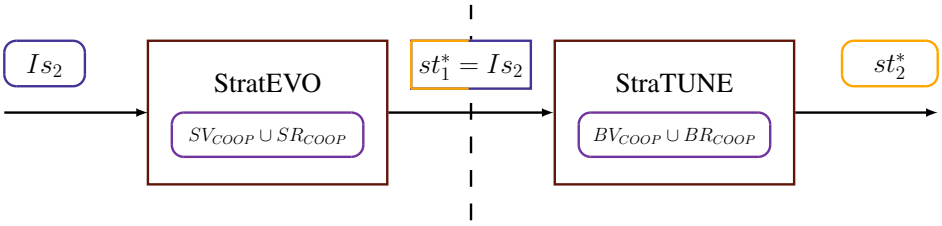



Figure 8.6 – SequencEVO: Off-line sequential cooperative scheme.

Figure 8.6 shows how SequencEVO works. From an initial strategy previously defined for the StratEVO engine (see Section 7.1.1), i.e., the Z3 default strategy or the StratGEN fixed skeleton strategy, then the evolutionary process is performed twice. First, StratEVO is in charge of generating an optimised structure strategy, to then calibrate the parameter vector of the best found strategy by means of StraTUNE.

Note that both StratEVO and StraTUNE remain practically unchanged, with an exception for the rules used by StratEVO. In SequencEVO, StratEVO is only used for applying Structural Modification rules, leaving Behavioural Variation rules $BV_{COOP}$ to be exclusively applied by StraTUNE.

## 8.4 HybridEVO, On-line Cooperative Engines

We modify the StratEVO engine, in order to include different on-line cooperative schemes in their evolutive process. These engines, named **HybridEVO**, are explained below. Let us remark that most StratEVO components remain unchanged, but the Evolutionary Algorithm Scheme (see Algorithm 5.1) functions are modified.

### 8.4.1 HybridEVO-1: On-line sequential cooperative engine

Our first engine, **HybridEVO-1**, is a modification of the StratEVO engine, shown in Figure 8.7, where behavioural modification rules are applied sequentially after using structural modification rules.

#### 8.4.1.1 Selection functions

Selection functions are in charge of choosing, at some point of the evolution process (i.e., iteration), a rule from $R_{COOP}$ and a set of individuals to apply it in order to generate a new

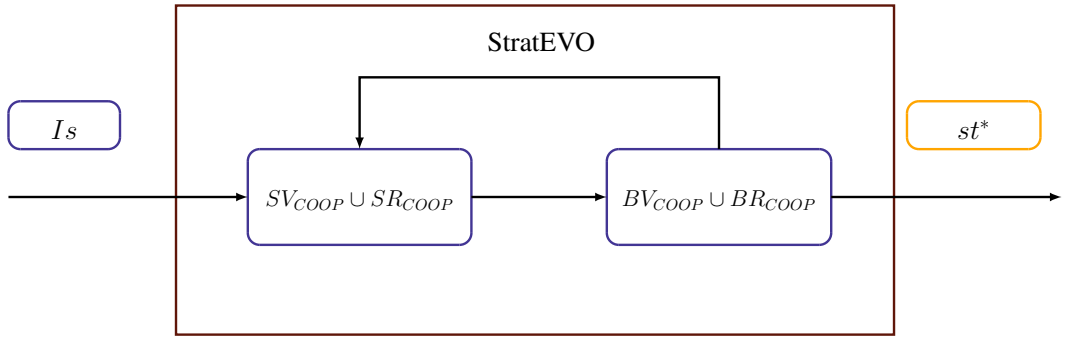



Figure 8.7 – HybridEVO-1: On-line sequential cooperative scheme.

strategy. Through this functions we assure to follow the sequential pattern previously explained.

Algorithm 8.1 shows how, through the $select_R$ function, rules from $R_{COOP}$ are selected in HybridEVO-1. First, a structural recombination rule ($r_r \in SR_{SEVO}$) is chosen. Then, in the following generation, it picks two variation rules ($r_v \in SV_{SEVO}$). Moreover, in the next iteration, it selects a behavioural recombination rule ($b_r \in BR_{COOP}$) and a behavioural variation rule ($b_v \in BV_{COOP}$).

**Algorithm 8.1:** HybridEVO-1 $select_R$ function

**Input:** a set of rules $R_{COOP}$
**Output:** A rule set $r$

1: **if** first iteration in turn **then**
2: $r \leftarrow \texttt{random}(SR_{COOP})$
3: **else if** second iteration in turn **then**
4: $r \leftarrow \texttt{random}(SV_{COOP}, 2)$
5: **else**
6: $r \leftarrow \texttt{random}(BR_{COOP})+\texttt{random}(BV_{COOP})$
7: **end if**
8: **return** $r$

As Algorithm 8.2 states, the $select_I$ function chooses a set of individuals to apply the selected rules. If the rule arity is two, a couple of individuals are selected by means of fitness using a classic tournament selection operator, such as in StratEVO. If the rule arity is one, the last member inserted in the population and its sibling are selected. Otherwise, the best from the two last individuals inserted in the population is selected.

**Algorithm 8.2:** HybridEVO-1 $select_I$ function

**Input:** an amount of individuals, $n$
a population, *population*
a fitness function *fitness*
**Output:** a set of individual $ind^n$

1: **if** $n = 2$ **then**
2: $ind^n \leftarrow$ `tournament_selection`(*population*,*fitness*,n)
3: **else if** $n = 1$ **then**
4: $last \leftarrow$ `last_inserted`(*population*)
5: $ind^n \leftarrow last, \texttt{sibling}(last)$
6: **else**
7: $mutated \leftarrow \texttt{apply}(last, \texttt{sibling}(last), r_v)$
8: $ind^n \leftarrow \texttt{best}(mutated)$
9: **end if**
10: **return** $ind^n$

Note that the arity of $r_r \in SR_{COOP}$ is always $ar(r_r) = 2$, then a tournament selection is exclusively performed to choose the parents for crossover operation. Also, $r_v \in SV_{COOP}$ arity is $ar(r_v) = 1$, therefore the variation rules will be exclusively applied to the offspring of a recombination. Moreover, $b_r \in BR_{COOP}$ and $b_v \in BV_{COOP}$ arity is always $ar(b_r + b_v) \geq 2$, therefore behavioural rules are applied separately over the best result obtained by means of the $r_v$ rules used in the previous generation.

#### 8.4.1.2 Insert function

Once a new strategy is generated, we need to insert them into the current population. The HybridEVO-1 *insert* function integrates new strategies by means of fitness in the actual generation.

Algorithm 8.3 shows how HybridEVO-1 inserts candidate strategies into the population. If new strategies were generated by means of recombination rules, the best offspring replaces the worst population member. Then, if those strategies are generated using mutation-driven structural rules, the best one replaces the worst individual different from the last inserted, if its fitness is better. Moreover, if those strategies were generated by behavioural rules (either recombination or variation rules), the best generated individual replaces the worst of the latest two inserted members if its fitness is better. Thus, a complete cycle of HybridEVO-1 is done every three generations.

**Algorithm 8.3:** HybridEVO-1 *insert* function

**input:** an individual set, $Ind'^{n'}$
a population, *population*
a fitness function *fitness*
**output:** a new *population* generation

```
1: new ← best(Ind'^n')
2: if first iteration in turn then
3:    worst ← worst(population,fitness)
4:    population ← replace(worst,new)
5: else if second iteration in turn then
6:    last ← last_inserted(population)
7:    worst ← worst(population-last,fitness)
8:    population ← replace(worst,new,fitness)
9: else
10:   last ← last_inserted(population,2)
11:   worst ← worst(last,fitness)
12:   population ← replace(worst,new,fitness)
13: end if
14: return population
```

### 8.4.2 HybridEVO-2: On-line segmented cooperative engine

Our second StratEVO variation, called **HybridEVO-2**, applies rules from the both Structural and Behavioural sets as shown in Figure 8.8. It main difference with regards to HybridEVO-1 is the way they are applied. As explained in Section 8.2.2.2, this engine segments the application of rules: first it focuses on Structural Modification rules for a defined amount of time, and then it changes to Behavioural Modification rules for another time period. As well as in HybridEVO-1, StratEVO main configuration remains unchanged with the exception of the evolutionary selection and insertion functions.

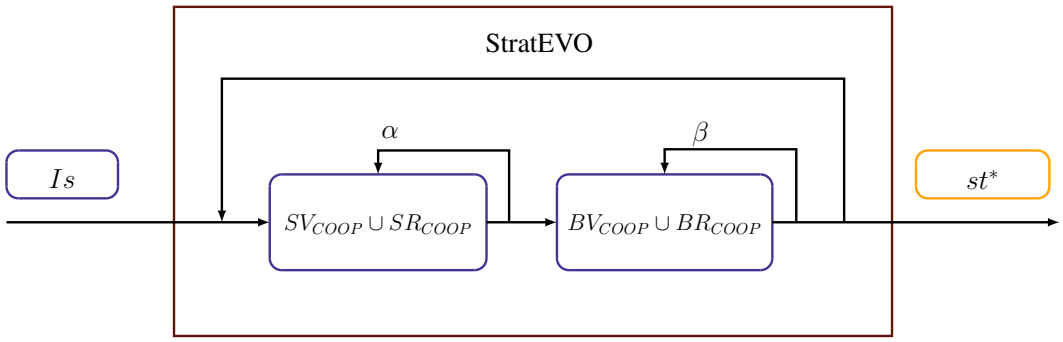



Figure 8.8 – HybridEVO-2: On-line segmented cooperative scheme.

#### 8.4.2.1 Selection functions

Selection functions are in charge of choosing rules from $R_{COOP}$ and the individuals related to them in order to generate new strategies. Through these functions, we assure to follow the segmented pattern previously explained.

HybridEVO-2 engine, as shown in Algorithm 8.4, has two limit values: $\alpha$ and $\beta$, which define how the *select*$_R$ function works. The $\alpha$ value indicates how many time rules from $SR_{COOP} \cup SV_{COOP}$ are performed, and after, $\beta$ value indicates the amount of times rule from $BR_{COOP} \cup BV_{COOP}$ are applied. Therefore, $\alpha + \beta$ is the total amount of rules applied in a complete cycle of a single segmented sequentialisation. Then, several cycles are iteratively applied until reaching ending criteria.

**Algorithm 8.4:** HybridEVO-2 *select*$_R$ function

**Input:** a set of rules $R_{COOP}$
**Output:** A rule set $r$

```
1: if generation ≤ α then
2:    if first iteration in turn then
3:       r ← random(SR_COOP)
4:    else if second iteration in turn then
5:       r ← random(SV_COOP, 2)
6:    end if
7: else
8:    r ← random(BR_COOP)+random(BV_COOP)
9: end if
10: return r
```

Note that in the structural modification stage, this function acts as in StratEVO, i.e., a structural recombination rule ($r_r \in SR_{COOP}$) is selected. In the next generation, two structural variation rules ($r_v \in SV_{COOP}$) are selected. If the engine process is in the behavioural modification stage, a behavioural recombination rule ($b_r \in BR_{COOP}$) and one behavioural variation rule ($b_v \in BV_{COOP}$) are chosen.

As well as for rule selection, the *select*$_I$ function is also segmented. Algorithm 8.5 shows how this function proceeds. If the current segment corresponds to the application of Structural Modification rules, the individual selection acts equally as in original StratEVO engine. That is, if a structural recombination rule is chosen, it will be applied over two parents selected by means of classic tournament selection. Otherwise, mutation-based rules will be applied over the offspring of the last crossover of the population.

**Algorithm 8.5:** HybridEVO-2 $select_I$ function

**Input:** an amount of individuals, $n$
a population, *population*
a fitness function *fitness*
**Output:** a set of individual $ind^n$

1: **if** generation $\leq \alpha$ **then**
2: **if** first iteration in turn **then**
3: $ind^n \leftarrow$ `tournament_selection`(*population*,*fitness*,n)
4: **else**
5: $last \leftarrow$ `last_inserted`(*population*)
6: $ind^n \leftarrow last,$ `sibling`$(last)$
7: **end if**
8: **else if** $\alpha <$ generation $\leq \alpha + \beta$ **then**
9: $ind^n \leftarrow$ `best`(*population*)
10: **end if**
11: **return** $ind^n$

However, if the current segment corresponds to Behavioural Modification rules, the best individual of the population is selected to modify, separately, its parameter vector by means of both selected behavioural recombination and variation rules.

#### 8.4.2.2 Insert function

Once a new strategy is generated, we need to insert it into the current population. The HybridEVO-2 *insert* function integrates new strategies by means of fitness in the current generation.

The *insert* function in HybridEVO-2 (shown in Algorithm 8.6) is also segmented. If the engine is in the $\alpha$ stage, the function acts as in StratEVO. That is, the best result of a structural recombination rule replaces the worst individual of the population, and the best result obtained applying structural variation rules replaces the worst element of the population different from the last inserted, with regards to fitness. But, if the engine is in $\beta$ segment, the best result obtained of using both Behavioural Modification rules replaces the worst population member.

**Algorithm 8.6:** HybridEVO-2 *insert* function

**input:** an individual set, $Ind'^{n'}$
a population, *population*
a fitness function *fitness*
**output:** a new *population* generation

```
1: new ← best(Ind'^n')
2: if generation ≤ α then
3:    if first iteration in turn then
4:       worst ← worst(population,fitness)
5:       population ← replace(worst,new)
6:    else if second iteration in turn then
7:       last ← last_inserted(population)
8:       worst ← worst(population-last,fitness)
9:       population ← replace(worst,new,fitness)
10:   end if
11: else
12:   worst ← worst(population,fitness)
13:   population ← replace(worst,new)
14: end if
15: return population
```

### 8.4.3 HybridEVO-3: On-line adaptive cooperative engine

The last cooperative schema, named **HybridEVO-3** and shown in Figure 8.9, aims at choosing rules depending on their performances in an adaptive procedure. As for other on-line cooperative engines, StratEVO main configuration remains unchanged with the exception of the evolutionary selection and insertion functions.

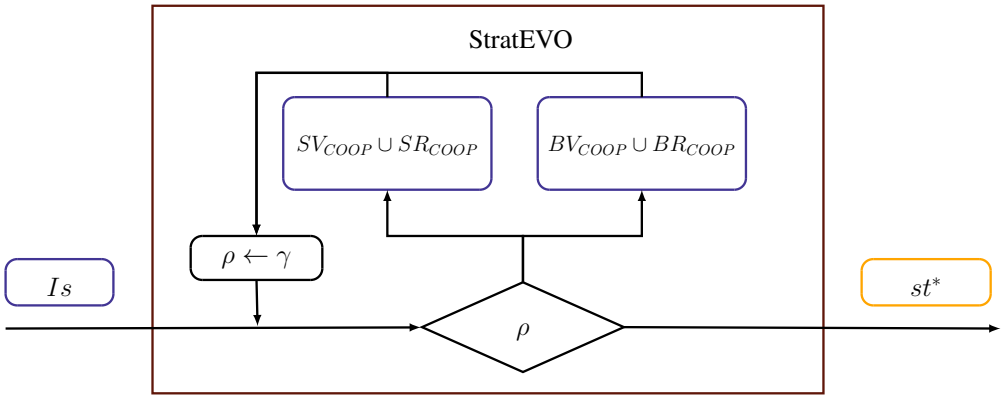



Figure 8.9 – HybridEVO-3: On-line adaptive cooperative scheme.

#### 8.4.3.1 Selection functions

Selection functions are in charge of choosing rules from $R_{COOP}$ and the individuals related to them, in order to generate new strategies. Through these functions, we assure to follow the adaptive pattern previously explained by using ratios whose values change depending on the success of the rule application.

As Algorithm 8.7 shows, HybridEVO-3 engine selects only one type of rules, depending on a flexible two-side ratio $\rho$. If $\rho$ probability is satisfied, HybridEVO-3 acts as StratEVO, i.e., a structural recombination rule ($r_r \in SR_{COOP}$) is selected. Then, in the next generation, two structural variations rules ($r_v \in SV_{COOP}$) are selected. But, if $\rho$ is not satisfied, a behavioural recombination rule ($b_r \in BR_{COOP}$) and one behavioural variation rule ($b_v \in BV_{COOP}$) are chosen.

**Algorithm 8.7:** HybridEVO-3 $select_R$ function

**Input:** a set of rules $R_{COOP}$
**Output:** A rule set $r$

```
1:  if random < ρ then
2:     if first iteration in turn then
3:        r ← random(SR_COOP)
4:     else if second iteration in turn then
5:        r ← random(SV_COOP, 2)
6:     end if
7:  else
8:     r ← random(BR_COOP)+random(BV_COOP)
9:  end if
10: return r
```

The $select_I$ function depends on the $\rho$ adaptive ratio. As shown in Algorithm 8.8, if $\rho$ succeeds, the function acts as in StratEVO, i.e., if a structural recombination rule is chosen, it will be applied over two parents selected by means of classic tournament selection. Otherwise, mutation-based rules will be applied over the offspring of the last crossover of the population. But, if $\rho$ does not succeeds, the best population individual will be selected to separately calibrate its parameter vector by means of a behavioural recombination rule and a behavioural variation rule.

**Algorithm 8.8:** HybridEVO-3 $select_I$ function

**Input:** an amount of individuals, $n$
a population, *population*
a fitness function *fitness*
**Output:** a set of individual $ind^n$

```
1: if random < ρ then
2:    if first iteration in turn then
3:       ind^n ← tournament_selection(population,fitness,n)
4:    else
5:       last ← last_inserted(population)
6:       ind^n ← last, sibling(last)
7:    end if
8: else
9:    ind^n ← best(population)
10: end if
11: return ind^n
```

#### 8.4.3.2 Insert function

Once a new strategy is generated, we need to insert them into the current population. The HybridEVO-3 *insert* function (as shown in Algorithm 8.9) integrates new strategies by means of fitness in the actual generation.

The insertion of new individuals in the population, also depends on the $\rho$ ratio. If the ratio succeeds, the insert function works as in StratEVO, i.e., the best result of a structural recombination rule replaces the worst individual of the population, and the best result obtained by applying structural variation rules replaces the worst element of the population (different from the last inserted) with regards to fitness. But, if the ratio does not succeed, the best result obtained using both Behavioural Modification rules replaces the worst population member.

The insert function also includes the instructions for changing the values of the $\rho$ ratio. Thus, if a structural modification improves strategy quality, the ratio is incremented of $\gamma$. Otherwise, it is reduced of $\gamma$. The inverse occurs when behavioural modification rules are used, i.e., the ratio is decreased of $\gamma$ if these rules generate a better strategy. Otherwise, it is increased of $\gamma$.

Let us remark that while $\rho$ increases, the possibility of applying structural rules also increases. In the other hand, if $\rho$ decreases, the possibility of applying behavioural rules is bigger.

**Algorithm 8.9:** HybridEVO-3 *insert* function

**input:** an individual set, $Ind'^{n'}$
a population, *population*
a fitness function *fitness*
**output:** a new *population* generation

1: $new \leftarrow$ `best`($Ind'^{n'}$)
2: **if** random < $\rho$ **then**
3: **if** first iteration in turn **then**
4: $worst \leftarrow$ `worst`(*population*,*fitness*)
5: *population* $\leftarrow$ `replace`($worst$,$new$)
6: **else if** second iteration in turn **then**
7: *last* $\leftarrow$ `last_inserted`(*population*)
8: $worst \leftarrow$ `worst`(*population-last*,*fitness*)
9: *population* $\leftarrow$ `replace`($worst$,$new$,*fitness*)
10: **end if**
11: **if** $new =$`best`(*population*) **then**
12: $\rho \leftarrow \rho + \gamma$
13: **else**
14: $\rho \leftarrow \rho - \gamma$
15: **end if**
16: **else**
17: $worst \leftarrow$ `worst`(*population*,*fitness*)
18: *population* $\leftarrow$ `replace`($worst$,$new$)
19: **if** $new =$`best`(*population*) **then**
20: $\rho \leftarrow \rho - \gamma$
21: **else**
22: $\rho \leftarrow \rho + \gamma$
23: **end if**
24: **end if**
25: **return** *population*

## 8.5 Experimental results

After defining the background of all cooperative engines, we analyse the results of their best strategies obtained using two different initial strategies. Later, the difference of the learning processes between both version is discussed.

| Logic | Subset | Engine | ($Ltopi$[s], $Topi$[s]) (1,1) | | | (10,10) | | | (10,2400) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | solved | time[s] | total[s] | solved | time[s] | total[s] | solved | time[s] | total[s] |
| LIA | *known* | Z3 | 201 | 2.10 | 2.10 | 201 | 1.86 | 1.86 | 201 | 1.80 | 1.80 |
| | | *StratEVO*$_{Z3}$ | 201 | 1.38 | 1.38 | 201 | 1.37 | 1.37 | 201 | 1.37 | 1.37 |
| | | SequencEVO$_{FS}$ | **201** | **1.29** | **1.29** | 201 | 1.33 | 1.33 | 201 | 1.36 | 1.36 |
| | | SequencEVO$_{Z3}$ | 201 | 1.42 | 1.42 | 201 | 1.26 | 1.26 | 201 | 1.49 | 1.49 |
| | | **HybridEVO-1**$_{FS}$ | 201 | 1.35 | 1.35 | **201** | **1.17** | **1.17** | **201** | **1.31** | **1.31** |
| | | HybridEVO-1$_{Z3}$ | 201 | 1.44 | 1.44 | 201 | 1.35 | 1.35 | **201** | **1.31** | **1.31** |
| | | HybridEVO-2$_{FS}$ | 201 | 2.06 | 2.06 | 201 | 1.55 | 1.55 | 201 | 2.29 | 2.29 |
| | | HybridEVO-2$_{Z3}$ | 201 | 1.91 | 1.91 | 201 | 1.43 | 1.43 | 201 | 2.09 | 2.09 |
| | | HybridEVO-3$_{FS}$ | 201 | 1.74 | 1.74 | 201 | 1.70 | 1.70 | 201 | 1.66 | 1.66 |
| | | HybridEVO-3$_{Z3}$ | 201 | 1.66 | 1.66 | 201 | 1.39 | 1.39 | 201 | 1.75 | 1.75 |
| | *unknown* | Z3 | 180 | 5.89 | 14.89 | 182 | 8.09 | 78.09 | 185 | 2866.79 | 12466.79 |
| | | ***StratEVO***$_{Z3}$ | **189** | **4.51** | **4.51** | **189** | **4.47** | **4.47** | **189** | **4.47** | **4.47** |
| | | **SequencEVO**$_{FS}$ | 189 | 7.85 | 7.85 | **189** | **4.47** | **4.47** | 189 | 6.20 | 6.20 |
| | | SequencEVO$_{Z3}$ | 189 | 6.71 | 6.71 | 189 | 5.12 | 5.12 | 189 | 6.87 | 6.87 |
| | | HybridEVO-1$_{FS}$ | 189 | 5.69 | 5.69 | 189 | 5.07 | 5.07 | 189 | 5.64 | 5.64 |
| | | HybridEVO-1$_{Z3}$ | 189 | 5.44 | 5.44 | 189 | 5.19 | 5.19 | 189 | 5.44 | 5.44 |
| | | HybridEVO-2$_{FS}$ | 189 | 10.25 | 10.25 | 189 | 5.13 | 5.13 | 189 | 11.42 | 11.42 |
| | | HybridEVO-2$_{Z3}$ | 189 | 10.49 | 10.49 | 189 | 5.09 | 5.09 | 189 | 11.15 | 11.15 |
| | | HybridEVO-3$_{FS}$ | 189 | 6.04 | 6.04 | 189 | 5.48 | 5.48 | 189 | 6.29 | 6.29 |
| | | HybridEVO-3$_{Z3}$ | 189 | 6.22 | 6.22 | 189 | 5.38 | 5.38 | 189 | 6.16 | 6.16 |
| LRA | *known* | Z3 | 331 | 6.69 | 14.69 | 333 | 10.83 | 70.83 | 337 | 491.00 | 5291.00 |
| | | ***StratEVO***$_{Z3}$ | **333** | **6.96** | **12.96** | 337 | 23.94 | 43.94 | **339** | **193.25** | **193.25** |
| | | SequencEVO$_{FS}$ | 333 | 9.13 | 15.13 | 337 | 24.72 | 44.72 | 339 | 206.10 | 206.10 |
| | | SequencEVO$_{Z3}$ | 328 | 7.52 | 18.52 | 337 | 28.98 | 48.98 | 339 | 223.24 | 223.24 |
| | | HybridEVO-1$_{FS}$ | 330 | 7.04 | 16.04 | 337 | 31.76 | 51.76 | 339 | 197.81 | 197.81 |
| | | HybridEVO-1$_{Z3}$ | 331 | 7.72 | 15.72 | **338** | **40.37** | **50.37** | 339 | 278.93 | 278.93 |
| | | HybridEVO-2$_{FS}$ | 329 | 8.74 | 18.74 | 337 | 31.92 | 51.92 | 339 | 269.34 | 269.34 |
| | | HybridEVO-2$_{Z3}$ | 329 | 8.29 | 18.29 | 337 | 27.38 | 47.38 | 339 | 385.97 | 385.97 |
| | | HybridEVO-3$_{FS}$ | 327 | 4.06 | 16.06 | 337 | 34.82 | 54.82 | 337 | 33.39 | 4833.39 |
| | | HybridEVO-3$_{Z3}$ | 331 | 8.03 | 16.03 | 337 | 31.07 | 51.07 | 336 | 22.91 | 7222.91 |
| | *unknown* | Z3 | 225 | 12.58 | 69.58 | 236 | 63.18 | 523.18 | 247 | 7324.12 | 91324.12 |
| | | *StratEVO*$_{FS}$ | **231** | **16.50** | **67.50** | 250 | 126.56 | 446.56 | 265 | 7635.47 | 48435.47 |
| | | **SequencEVO**$_{FS}$ | 212 | 15.93 | 85.93 | **252** | **145.56** | **445.56** | **266** | **6951.99** | **45351.99** |
| | | SequencEVO$^{10}_{FS}$ | 208 | 14.18 | 88.18 | 216 | 43.34 | 703.34 | 219 | 136.22 | 151336.22 |
| | | SequencEVO$_{Z3}$ | 200 | 7.10 | 89.10 | 252 | 257.45 | 557.45 | 265 | 5715.46 | 46515.46 |
| | | SequencEVO$^{10}_{Z3}$ | 224 | 27.91 | 85.91 | 248 | 105.51 | 445.51 | 262 | 9212.89 | 57212.89 |
| | | HybridEVO-1$_{FS}$ | 201 | 9.91 | 90.91 | 250 | 170.96 | 490.96 | 264 | 6477.16 | 49677.16 |
| | | HybridEVO-1$_{Z3}$ | 199 | 5.43 | 88.43 | 249 | 153.37 | 483.37 | 264 | 8472.22 | 52672.22 |
| | | HybridEVO-2$_{FS}$ | 199 | 8.81 | 91.81 | 250 | 140.07 | 460.07 | 263 | 9003.18 | 54603.18 |
| | | HybridEVO-2$_{Z3}$ | 199 | 10.76 | 93.76 | 251 | 211.71 | 521.71 | 266 | 9061.85 | 47461.85 |
| | | HybridEVO-3$_{FS}$ | 197 | 10.11 | 95.11 | 250 | 255.40 | 575.40 | 264 | 7563.36 | 50763.36 |
| | | HybridEVO-3$_{Z3}$ | 201 | 7.20 | 88.20 | 249 | 237.07 | 567.07 | 265 | 7357.86 | 48157.86 |

Table 8.1 – SMT-LIB Benchmarks: Solving LIA and LRA logic instances set using strategies generated by different cooperative engines.

| Logic | Subset | Engine | ($Ltopi$[s], $Topi$[s]) (1,1) solved | (1,1) time[s] | (1,1) total[s] | (10,10) solved | (10,10) time[s] | (10,10) total[s] | (10,2400) solved | (10,2400) time[s] | (10,2400) total[s] |
|---|---|---|---|---|---|---|---|---|---|---|---|
| QF_LIA | *known* | *Z3* | 2879 | 637.01 | 3597.01 | 4102 | 5319.20 | 22689.20 | 5617 | 126486.39 | 659286.39 |
| | | SequencEVO$^{10}_{FS}$ | 2826 | 637.19 | 3650.19 | 3973 | 4651.05 | 23311.05 | 5615 | 148255.40 | 685855.40 |
| | | SequencEVO$^{10}_{Z3}$ | 2382 | 702.07 | 4159.07 | 4097 | 6948.00 | 24368.00 | 5598 | 115216.42 | 693616.42 |
| | | HybridEVO-1$_{FS}$ | **2893** | **625.46** | **3571.46** | 3984 | 4656.49 | 23206.49 | 5615 | 145628.75 | 683228.75 |
| | | **HybridEVO-1$_{Z3}$** | 2828 | 625.46 | 3636.46 | **4209** | **4656.49** | **20956.49** | **5630** | **131832.62** | **633432.62** |
| | | HybridEVO-2$_{FS}$ | 2681 | 694.56 | 3852.56 | 3964 | 5019.21 | 23769.21 | 5599 | 180485.94 | 756485.94 |
| | | HybridEVO-2$_{Z3}$ | 2856 | 741.61 | 3724.61 | 4168 | 5534.55 | 22244.55 | 5578 | 140204.41 | 766604.41 |
| | | HybridEVO-3$_{FS}$ | 2482 | 529.45 | 3886.45 | 3998 | 6644.98 | 25054.98 | 5617 | 141935.57 | 674735.57 |
| | | HybridEVO-3$_{Z3}$ | 2889 | 612.70 | 3562.70 | 4102 | 5314.11 | 22684.11 | 5605 | 148997.18 | 710597.18 |
| | *unknown* | Z3 | 81 | 37.82 | 258.82 | 110 | 200.39 | 2120.39 | 130 | 13230.82 | 426030.82 |
| | | *StratEVO$_{Z3}$* | 197 | 33.25 | 138.25 | 198 | 41.21 | 1081.21 | 210 | 13617.78 | 234417.78 |
| | | SequencEVO$_{FS}$ | 81 | 39.43 | 260.43 | 109 | 204.39 | 2134.39 | 128 | 16459.01 | 434059.01 |
| | | SequencEVO$^{10}_{FS}$ | 74 | 44.27 | 272.27 | 99 | 195.26 | 2225.26 | 126 | 13736.21 | 436136.21 |
| | | SequencEVO$_{Z3}$ | **198** | **15.46** | **119.46** | 203 | 41.21 | 1031.21 | 210 | 15370.77 | 236170.77 |
| | | **SequencEVO$^{10}_{Z3}$** | 155 | 85.63 | 232.63 | **206** | **339.17** | **1299.17** | **214** | **21722.32** | **232922.32** |
| | | HybridEVO-1$_{FS}$ | 76 | 39.61 | 265.61 | 109 | 199.37 | 2129.37 | 131 | 16909.19 | 427309.19 |
| | | HybridEVO-1$_{Z3}$ | 159 | 74.07 | 217.07 | 199 | 157.69 | 1187.69 | 209 | 14872.01 | 238072.01 |
| | | HybridEVO-2$_{FS}$ | 77 | 42.19 | 267.19 | 109 | 201.76 | 2131.76 | 127 | 15714.89 | 435714.89 |
| | | HybridEVO-2$_{Z3}$ | 149 | 64.70 | 217.70 | 199 | 163.88 | 1193.88 | 206 | 6088.18 | 236488.18 |
| | | HybridEVO-3$_{FS}$ | 81 | 43.56 | 264.56 | 109 | 188.02 | 2118.02 | 124 | 12818.97 | 440018.97 |
| | | HybridEVO-3$_{Z3}$ | 83 | 41.50 | 260.50 | 199 | 342.72 | 1372.72 | 211 | 16431.27 | 234831.27 |
| QF_LRA | *known* | Z3 | 1054 | 71.62 | 643.62 | 1173 | 494.45 | 5024.45 | 1530 | 116198.57 | 346598.57 |
| | | ***StratEVO$^{10}_{FS}$*** | **1108** | **98.93** | **616.93** | **1288** | **881.52** | **4261.52** | **1583** | **44576.55** | **147845.73** |
| | | SequencEVO$^{10}_{FS}$ | 1095 | 92.48 | 623.48 | 1264 | 828.97 | 4448.87 | 1575 | 70625.39 | 193052.39 |
| | | SequencEVO$^{10}_{Z3}$ | 1097 | 79.72 | 608.72 | 1253 | 796.49 | 4526.29 | 1578 | 76233.34 | 191433.34 |
| | | HybridEVO-1$_{FS}$ | 1047 | 95.80 | 674.80 | 1255 | 994.32 | 4704.32 | 1551 | 56922.05 | 236922.05 |
| | | HybridEVO-1$_{Z3}$ | 1093 | 91.99 | 624.99 | 1240 | 746.86 | 4606.86 | 1577 | 57974.16 | 175574.16 |
| | | HybridEVO-2$_{FS}$ | 988 | 79.40 | 717.40 | 1246 | 1116.63 | 4916.63 | 1553 | 94832.95 | 270032.95 |
| | | HybridEVO-2$_{Z3}$ | 1080 | 99.74 | 645.74 | 1232 | 779.18 | 4719.18 | 1570 | 74811.89 | 209211.89 |
| | | HybridEVO-3$_{FS}$ | 1032 | 57.72 | 651.72 | 1254 | 954.03 | 4674.03 | 1556 | 49519.42 | 217519.42 |
| | | HybridEVO-3$_{Z3}$ | 1102 | 94.30 | 618.30 | 1259 | 768.26 | 4438.26 | 1571 | 52974.65 | 184974.65 |
| | *unknown* | Z3 | 0 | 0.00 | 56.00 | 0 | 0.00 | 560.00 | 2 | 2886.58 | 132486.58 |
| | | ***StratGEN*** | **5** | **1.88** | **52.88** | **18** | **90.22** | **470.22** | **50** | **5793.71** | **20193.71** |
| | | SequencEVO$_{FS}$ | 0 | 0.00 | 56.00 | 4 | 33.43 | 553.43 | 23 | 6544.05 | 85744.05 |
| | | SequencEVO$_{Z3}$ | 0 | 0.00 | 56.00 | 1 | 4.95 | 554.95 | 31 | 19865.59 | 79865.59 |
| | | HybridEVO-1$_{FS}$ | 0 | 0.00 | 56.00 | 5 | 30.00 | 540.00 | 14 | 4600.29 | 105400.29 |
| | | HybridEVO-1$_{Z3}$ | 0 | 0.00 | 56.00 | 1 | 3.91 | 553.91 | 3 | 3085.20 | 45090.02 |
| | | HybridEVO-2$_{FS}$ | 0 | 0.00 | 56.00 | 6 | 44.16 | 544.16 | 6 | 1477.97 | 121477.97 |
| | | HybridEVO-2$_{Z3}$ | 0 | 0.00 | 56.00 | 0 | 0.00 | 560.00 | 0 | 0.00 | 134400.00 |
| | | HybridEVO-3$_{FS}$ | 0 | 0.00 | 56.00 | 14 | 38.50 | 458.50 | 20 | 4617.79 | 91017.79 |
| | | HybridEVO-3$_{Z3}$ | 0 | 0.00 | 56.00 | 1 | 7.35 | 557.35 | 36 | 22096.20 | 70096.20 |

Table 8.2 – SMT-LIB Benchmarks: Solving QF_LIA and QF_LRA instances set using strategies generated by different cooperative engines.

### 8.5.1 Performance highlights

As shown in Table 8.1 and Table 8.2, we analyze the performance of the Z3 solver over the selected logics by using its default strategy and the best strategies generated by our cooperative engines. We also include the best performer engine so far from previous chapters. Let us remark the following:

1. In each logic subset, i.e., *known* and *unknown*, the engine which triggers Z3 best global result between all execution scenarios will be highlighted in **bold**.
2. The best engine from previous chapter will be highlighted in *italic*.
3. The est result in each execution scenario will also be highlighted in **bold**.
4. As engines are based on StratEVO, they share the two available options for initial strategy. We use Z3 to denote the use of the default Z3 strategy, or FS if the fixed skeleton strategy is selected. This will be noted as subscript of the engine, e.g., HybridEVO-$1_{Z3}$.
5. SequencEVO engine can use a reduced learning set ($Lspct = 10\%$) to address a logic subset, it will be denoted with the number 10 as superscript of the engine, e.g., SequencEVO$^{10}_{FS}$.

Note that we mention engines performance referring to the effects of their best generated strategy for the Z3 solver. The same principle applies to the engines that *outperform* Z3.

#### 8.5.1.1 Z3 improvements

Engines including cooperative schemes win in 5 of 8 scenarios, i.e., 13 of 24 cases, being slightly better than the best found so far, specially in learning conditions, i.e., $Topi = Ltopi = 10$ seconds. However, the success in the learning phase does not mean an improvement under SMT-COMP conditions ($Topi = 2400$ seconds) as happens in the logics which are completely solved: `LIA` *unknown* and `LRA` *known*. In these scenarios, cooperative engines are outperformed by StratEVO in terms of execution time. Moreover, cooperative engines could not match the best solver performance in the whole `QF_LRA` logic set.

Despite those cases, the use of the cooperative schemes reaches some interesting milestones for the hardest set of instances. HybridEVO-1 is the first engine which outperforms Z3 default strategies in all scenarios and cases, including being the first solving process for the `QF_LIA` logic set.

Major achievement includes:

— HybridEVO-1 outperforms Z3 default performance in `QF_LIA` *known*, solving around 5% of its unaddressed instances.

— SequencEVO outperforms the best engine in `QF_LIA` *unknown*, addressing around 5% of the unsolved instances and reducing of 1% the execution time.

— SequencEVO outperforms the best engine in LRA *unknown* by solving around 5% of unsolved instances and shrinking 7% of the processing time.

#### 8.5.1.2 Initial Strategy factor

The selection of the initial strategy still plays a great role in engine performance. But, these differences are dramatically reduced for engines using cooperative schemes, with some few exceptions as the QF_LIA instance set. We also observe that the improvement ratio of some generated strategies is not reproducible, and it drastically varies, when they are used under SMT-COMP rules.

No major difference between the best result obtained using Z3 default and StratGEN fixed skeleton strategies could be seen for LIA and LRA benchmarks. However, the use of Z3 default strategy as starting point leads to the best engines performances for QF_LIA logic, specially in the *unknown* subset with a difference of 67% more solved instances. In the QF_LRA logic, the engines perform slightly better in the learning phase by using fixed skeleton strategy, but this behaviour is not reproducible in SMT-COMP conditions. The same behaviour is also seen in QF_LIA *known* with regards to the Z3 default strategy, and it was previously stated as an important fact to improve the engine strategy generation process.

#### 8.5.1.3 Off-line vs On-line cooperative schemes

Both types of engines have different peaks of performance depending on the given SMT logic. Thus, SequencEVO slightly outperforms HybridEVO engines in four logics subsets: LIA *unknown*, LRA *unknown*, QF_LIA *unknown* and QF_LRA *known*. Meanwhile, HybridEVO engines slightly outperform SequencEVO in the remaining four logic sets. However, the behaviour between both types of engines are very similar, with the exception of QF_LIA *known* in favor of HybridEVO.

In spite of this analysis, note that SequencEVO completely executes two different engines (one for each type of rules), i.e., it has a double learning budget time: $2+2 = 4$ days. Therefore, HybridEVO engines match SequencEVO performance in half time, by including both rules in the execution environment of a single engine.

With regards to engines based on on-line cooperative schemes, HybridEVO-1 has better performance in 5 of 8 scenarios, followed by HybridEVO-3 obtaining the best results in 3 of 8. Hence, HybridEVO-2 is the worst performer between this types of engines.

### 8.5.2 Learning process

As well as all designed engines, they performance rely on a learning phase defined in our framework for automated generation of strategies. Thus, we explain here how this phase sup-

ports obtained results.

#### 8.5.2.1 Learning sample size

As cooperative engines are based on StratEVO, they work over different learning samples to generate efficient and complex strategies. SequencEVO uses the same samples as StratEVO, but the HybridEVO engines uses only one type of samples for each logic. For more information see Section 5.4.3.

As in StratEVO, using a learning sample of 10% of the whole benchmark is very useful for generating optimised strategies by means of less consumption of computational resources. Note that, in the `QF_LIA` *known* subset and the `QF_LRA` set, HybridEVO engines use exclusively a reduced learning sample. In the `QF_LIA` *known* logic, it allows to outperform for the first time the Z3 default performance, meanwhile in `QF_LRA` *known* it generates slightly worse alternatives with a difference of less than 1% of instances. With regards to SequencEVO, the use of a reduced learning sample allows to outperform the best result so far in `QF_LIA` *unknown*.

Consequently, the initial strategy has demonstrated, through all designed engines, that it is a relevant component of algorithm design that allows to generate more complex and sophisticated strategies which outperform the default performance of the Z3 solver.

#### 8.5.2.2 Learning Variability

Our cooperative engines, based on StratEVO, have a stochastic process for selecting rules. Thus, the engine learning phase was executed several times with different random seeds values.

As shown in Figure 8.10, SequencEVO generation of strategies is less sensitive to random seed values. On the other hand, HybridEVO engines have mixed variability depending on the selected initial strategy $Is$. Thus, if the fixed skeleton strategy is chosen, the performance is considerably variable. Meanwhile, if the Z3 default strategy is picked, the engine performance is more stable and similar to SequencEVO (with exception for `QF_LIA` *unknown*). Note that the SequencEVO stability is heavily influenced by its total learning time of four days. However, HybridEVO can obtain similar stable performance in half of this time.

With regards to the initial strategy, we observe a huge difference in the learning phase between the Z3 default and the fixed skeleton strategies. Hence, if the fixed skeleton strategy is chosen, the generated strategy have lower quality.

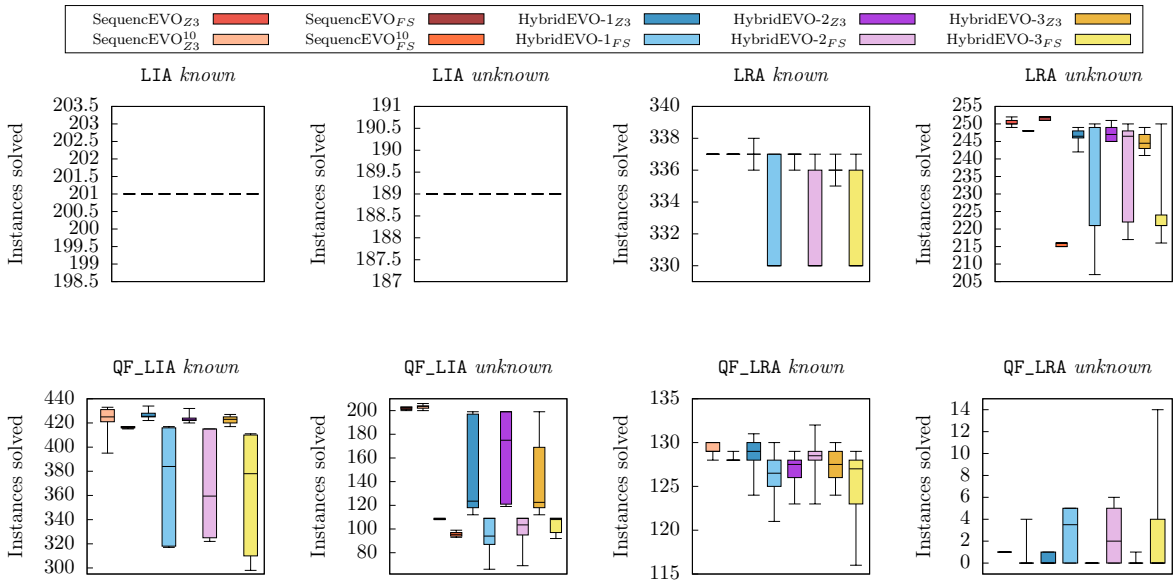



Figure 8.10 – Cooperative engines learning variability in the strategy generation process according to SMT logics with $Ltopi = 10$ seconds.

However, the differences between off-line and on-line collaborative schemes are not very significant. Table 8.3 summarises a set of Two-Tailed Student T Test for means of paired samples, with significance level of $\alpha = 0.05$. They measure the average performance of the two best engines based on off-line and on-line cooperative schemes (one of each). These tests stand statistically significant difference between the performance of both alternatives, being executed ten times with different random seed values. Once again, our tests mainly use data of instance solving. However, when both selected engines are completely tied in this metric, the analysis is done using time execution information.

We could check that no statistical significant difference exists in 5 of 8 cases. Meanwhile, the significative differences are given in *unknown* subsets of `LIA`, `LRA` and `QF_LIA` logics.

| Logic | Subset | Type | Diff. Mean | RMSD | $t$ value | $p$ value | better | status |
|---|---|---|---|---|---|---|---|---|
| `LIA` | *known* | time | -0.10 | 0.42 | -1.38 | **0.199712** | HybridEVO-1$_{FS}$ | ✗ |
| | *unknown* | time | -0.69 | 0.97 | -6.64 | **0.000095** | HybridEVO-1$_{FS}$ | ✓ |
| `LRA` | *known* | instances | -0.10 | 2.90 | -0.56 | **0.591051** | SequencEVO$_{FS}$ | ✗ |
| | *unknown* | instances | -4.40 | 42.40 | -6.41 | **0.000124** | SequencEVO$_{FS}$ | ✓ |
| `QF_LIA` | *known* | instances | 3.60 | 1816.40 | 0.80 | **0.443575** | HybridEVO-1$_{Z3}$ | ✗ |
| | *unknown* | instances | -40.00 | 12860.00 | -3.35 | **0.008574** | SequenceEVO$^{10}_{Z3}$ | ✓ |
| `QF_LRA` | *known* | instances | -0.50 | 34.50 | -0.81 | **0.440158** | HybridEVO-1$_{Z3}$ | ✗ |
| | *unknown* | instances | 1.60 | 70.40 | 1.81 | **0.103888** | HybridEVO-2$_{FS}$ | ✗ |

Table 8.3 – Student T Test: Statistical significance between best off-line and on-line cooperative engines in the learning phase of each logic, with level $\alpha = 0.05$.

## 8.6 Conclusions

In this chapter we presented several engines based on StratEVO: the SequencEVO and the HybridEVO engines. They add different cooperative schemes for integrating two types of rules: Structural and Behavioural modification rules. These engines are capable to evolve and outperform Z3 default configuration performances in all scenarios. Also, some of them outperform the currently best engine in many scenarios.

The most relevant improvement occurs in the hardest SMT logic selected for this thesis (`QF_LRA`) where through the use of cooperative schemes better results are achieved. Specially in the *known* subset, where for the first time, the Z3 default configuration is outperformed.

Moreover, the learning phase of these new engines are analysed. The variability between them is slightly different, with several engines performing at same level in several logics, as `LIA`, `LRA` and `QF_LRA` logics.

The next step of this work is to propagate the behaviour generated strategies to different execution contexts, in order to improve their overall efficiency.

# 9

# Expanding Strategies Semantics

In this chapter, we address the semantics gap obtained on optimised strategies when they are used under SMT-COMP competition rules. Through the definition and use of a new set of rules, we try to expand the behaviour of generated strategies to any execution scenario.

## 9.1 ExpandEVO: Expanding Time Semantics

An observed behaviour in several strategies generated by different engines is the inability to sustain its performance in scenarios with greater time-out per instance ($Topi$) budget. Let us remark that (as explained in Section 5.4.4) our engines generate strategies using a defined time-out per instance ($Ltopi = Topi$) of ten seconds in the learning phase. Meanwhile, the validation is done over SMT-COMP rules with forty minutes limit per instance. Note that the difference of execution time is about *two thousand three hundred and ninety seconds*.

### 9.1.1 Strategy Semantics

Based on classic semantics concepts in Evolutionary Computing (shown in Section 2.1.2.2), we define the concept of **semantics of the strategy**:

*The ability of a strategy to performs proportionally*
*equal in different execution scenarios*

That is to say, it generates the same performance level (i.e., fitness variation proportion) under different time-out per instance cases. We also we refer to this as the *meaning* of a strategy.

**Example 9.1** Let be $S_{10}$, shown in Figure 9.1, a strategy generated by means of an engine using a learning time-out per instance of 10 seconds. Under learning conditions, i.e., $Ltopi = Topi = 10$ seconds, this strategy equally distributes the available time between all solving tactics: 2.5 seconds for each solver (between lines 9 and 15), corresponding to 25% of the global time. When using $S_{10}$ under SMT-COMP conditions, i.e., 2400 seconds, this balanced distribution, as well as any distribution, will be dramatically modified with regards to the available execution time. Thus, the semantic of the strategy is also modified.

```
1  (and-then
2          simplify
3          (using-params ctx-simplify :max_depth 30 :max_steps 5000000)
4          solve-eqs
5          elim-uncnstr
6          propagate-ineqs
7          split-clause
8          (or-else
9                  (try-for sat 2500)
10                 (try-for smt 2500)
11                 (try-for qe-sat 2500)
12                 (and-then
13                         qe
14                         smt
15                 )
16         )
17
18 )
```

Figure 9.1 – Example of generated strategy, $S_{10}$, with $Ltopi = 10$ seconds.

Note that solvers with explicitly written time limits, i.e., defined by using `try-for` function, will not change their values, but the proportion drastically changes with regards to the time-out per instance ($Topi$). Meanwhile, solving tactics with no `try-for` functions, will change both time limit and proportion with regard to global time execution ($Topi$). Thus, for $S_{10}$ under SMT-COMP limits, the first three solvers (lines 9 to 11) still having a time budget of 2.5 seconds, but each of these limits corresponds to 0.1% of total time. On the other hand, last solving tactic (lines 12 to 15) has a time limit of 2932.5 seconds, i.e., 99.7% of the total time limit. ♢

Naturally, a question arises: ***How to adapt a strategy to different execution scenarios?***. We define **ExpandEVO**, a modification of the best learning performance engines which includes a new set of rules that handle time limit components to expand the semantics of strategy obtained in the learning phase to different execution scenarios.

### 9.1.2 Rules

In order to modify time components of a generated strategy, a set of expansion rules, $ER_{EEVO}$ is defined to it with regards to semantics aspects (i.e., time management). These rules

are applied after generating an optimised strategies. Indeed, our framework can also include modification rules acting outside the learning phase. Thus, this set of rules includes Structural and Behavioural variation rules, which are used depending on the method to expand the semantics, i.e., the amount of time in the strategy. We define and use two simple methods: direct proportion and classic linear regression models.

#### 9.1.2.1 Direct Proportion

The first alternative to expand the semantics of a strategy, is a simple and straightforward method: a proportionally mapping for time components into the new time limits (i.e., execution scenario). Thus, the original time proportion of the generated strategy will be hold in any scenario. Hence, we define the following rule:

— Proportional Expanded Time-out: it allows to proportionally expand a tactic timeout into another execution scenario. Thus, let $t$ be the $k^{\text{th}}$ component of the parameter vector $\pi$ of a strategy $st$, denoted as $\pi_k = t$, which corresponds to the solver time limit of a tactic.

$$PT_{EEVO} : \texttt{try-for}_{/2}(st, t) \rightarrow \texttt{try-for}_{/2}(st, t') \left\{ \frac{t}{Ltopi} = \frac{t'}{Topi} \right\}$$

where $t \propto Ltopi$ and $t' \propto Topi$. Note that the parameter vector $\pi$ is turned into $\pi'$, where $\texttt{compatible}(\pi, \pi')$ is fulfilled because $t \neq t'$, i.e, unique difference between $\pi$ and $\pi'$ is $\pi_k \neq \pi'_k$.

#### 9.1.2.2 Regression Models

The second alternative is to expand the semantics of the strategy by using different simple regression models for estimate time limit values according to a defined time-out per instance ($Topi$). Of course, these models need a sample of optimised strategies for several $Topi$ values.

— Regression Expanded Time-out: it allows to expand a tactic timeout into another execution scenario by using a regression model function. Thus, let $t$ be the $k^{\text{th}}$ component of the parameter vector $\pi$ of a strategy $st$, denoted as $\pi_k = t$, which corresponds to the solver time limit of a tactic.

$$RT_{EEVO} : \texttt{try-for}_{/2}(st, t) \rightarrow \texttt{try-for}_{/2}(st, t') \; \{f(Topi) = t'\}$$

where $f$ is the regression function to obtain the new value of $\texttt{try-for}_{/2}$ limit with regards to the global time-out per instance ($Topi$). Note that the parameter vector $\pi$ is turned into $\pi'$, where $\texttt{compatible}(\pi, \pi')$ is fulfilled because $t \neq t'$, i.e, unique difference between $\pi$ and $\pi'$ is $\pi_k \neq \pi'_k$.

#### 9.1.2.2.1 Regression functions

Let be $st \in$ `Strat` a strategy containing $m$ tactics related to `try-for` modifier functions. To obtain the values of their time limits, we define a classic regression model function as:

$$\hat{f}(t) = \hat{a}t + \hat{b}$$

where $t$ is the independent input variable corresponding to the time-out per instance ($Topi$) to use with $st$, and $f(t)$ the value of the dependent variable representing the time limit parameter component. Indeed, this function is a vector of $m$ regression functions for each `try-for` in strategy $st$. Thus the regression functions and its components are:

$$\begin{aligned}\hat{f}(t) &= [f_1(t), f_2(t), \ldots, f_m(t)] \\ \hat{a} &= [a_1, a_2, \ldots, a_m] \\ \hat{b} &= [b_1, b_2, \ldots, b_m]\end{aligned}$$

where:

$$f_i(t) = a_i t + b_i; \forall i \in \{1, \ldots, m\}$$

Through the use of diverse mapping, we define the following linear regression functions $\forall i \in \{1, \ldots, m\}$:

— Simple linear regression: It relates the input variable and its output value in a direct linear combination.

$$f_i(t) = a_i t + b_i$$

— Inverted regression: It relates the input variable and its output value in a inverted linear combination.

$$f_i(t) = \frac{a_i}{t} + b_i$$

— Exponential regression: It relates the independent variable with the dependent value in a exponential power relation.

$$f_i(t) = e^{a_i t}\, b_i$$

Through the use of logarithm operations, this combination could be linearly expressed as:

$$f'_i(t) = a_i t + b'_i$$

where $f'_i(t) = ln(f_i(t))$ and $b'_i = ln(b_i)$.

— Power regression: It relates the independent variable with the dependent value in a power

relation.

$$f_i(t) = t^{a_i}\ b_i$$

Through the use of logarithmic properties, this combination could be expressed as:

$$f'_i(t) = a_i t' + b'_i$$

where $f'_i(t) = ln(f_i(t))$, $t' = ln(t)$, and $b'_i = ln(b_i)$.

— Polynomial regression: It relates input variable with the dependent output by means of a polynomial relation.

$$f_i(t) = a_{i0}t^0 + a_{i1}t^1 + \cdots + a_{in}t^n$$

Note that the extension of the length of the polynomial depends on the degree used. While higher the polynomial degree, the greater is the fit to the sample. We used quadratic and fifth degree polynomials.

Note that, we need several pairs $(t, f(t))$, in order to obtain values of vectors $\hat{a}$ and $\hat{b}$. Also, despite of using non-linear methods, the regression models are still linear. This is because, the expressions on the right hand side of each $f_i(t)$ function are linear with regards to the parameters $a_i$ and $b_i$.

#### 9.1.2.3 Strategy Semantics Equivalence

From the three type of basic tactics (see Section 4.2.2), i.e., terminal symbols of a strategy, only *solvers* are affected by time configuration. Remaining components, both *heuristics* and *probes*, are straightforward process whose time consumption is irrelevant with regards to the total time. Thus, we could generate semantically equivalent strategies by applying the following structural modification rules:

— Add time limit to solver tactics: Adds a specific time limit to solving tactics that does not have explicitly defined a `try-for` modifying function.

$$AT_{EEVO} : s \rightarrow \texttt{try-for}_{/2}(s, t)$$

— Delete time limit into non-solving tactics: Removes time configuration values in tactics that not belong to `Solver` set.

$$DT_{EEVO} : \texttt{try-for}_{/2}(st, t) \rightarrow st\ \{st \notin \texttt{Solver}\}$$

The idea is to attach the time configuration specifically to each basic solver tactic, even if it originally does not have an explicit `try-for` function defined. This implies to also remove

time components related to composed or not solving basic tactics. Then, strategy semantics can be easily expanded by means of behavioural modifications.

### 9.1.3 Engines

As previously explained, ExpandEVO extends designed engines by adding rules to expand it semantics. Next, we define a set engines to be expanded for the addressed SMT logics of this work.

| Subset | | *known* | *unknown* |
|---|---|---|---|
| Logics | LIA | HybridEVO-1$_{Z3}$ | StratEVO$_{Z3}$ |
| | LRA | StratEVO$_{Z3}$ | SequencEVO$_{FS}$ |
| | QF_LIA | HybridEVO-1$_{Z3}$ | SequencEVO$^{10}_{Z3}$ |
| | QF_LRA | StratEVO$^{10}_{FS}$ | StratEVO$_{FS}$ |
| | | Engines | |

Table 9.1 – Expanded Engines Examples: Best StratEVO based engines for selected instance set.

#### 9.1.3.1 Expanded Engines

We select the best StratEVO-based performers in each logic to expand its semantics. This is because, most of them have great performance in the learning phase, but some of them cannot hold the same performance level under SMT-COMP rules. Thus, others engines reach similar or superior performances, as shown in Chapter 7 and Chapter 8. Table 9.1 summarises the selected engines.

#### 9.1.3.2 Sample Engines

In order to apply the expansion rules based in regression models, we need to define a sample. To generate this sample, we use StraTUNE engine (see Section sec:StraTUNE) over fifty different time-out per instance ($Topi$) values. The $Topi$ values are equally distributed from values between one and one hundred twenty five seconds. Then, this sample represents a set of fifty optimised strategy configuration in different execution scenarios.

Of course, for this task, StraTUNE engine is constrained to work only over time limit configurations. Also, global time budget ($Ltb$) is set to half a day (twelve hours) and with a new iteration limit of one hundred generations. Moreover, learning sample size used is twenty five instances, with exception of QF_LRA *unknown* set which uses ten instances.

## 9.2 Results

Next, we present the performances of strategies obtained by selected engines after expanding its time configuration semantics.

### 9.2.1 Performance highlights

We analyze the performance of time processed strategies generated by the selected engines, and we compare with it original form as well as Z3 default configuration output. For these results, let us consider the following:

1. We focus on logics which are not completely addressed, thus we let aside `LIA` and `LRA` *known* sets.
2. We test them two scenarios: a reduced learning sample size (of 25 instances) and the whole instance set.
3. In each logic subset, i.e., *known* and *unknown*, the best engine global result between all execution scenarios will be highlighted in **bold**.
4. Selected engine for expansion will be highlighted in *italic*.
5. Best result in each execution scenario will also be highlighted in **bold**.

#### 9.2.1.1 Z3 improvements

##### 9.2.1.1.1 Adapting originally generated strategies

As shown in Table 9.2, the use of rules for time semantics expansion allows to improve `LRA` *unknown* and `QF_LIA` logic sets. In `LRA` *unknown* subset, the improvement is limited, solving one more instance than the best strategy configuration. This improvement imply a time effort increase of 4.4%. However, the improvements on `QF_LIA` logic set are remarkable. In the *known* subset, the use of expansion rules outperforms StratEVO by solving the double of its improvements. It also reduce in 2.5% of execution time. For *unknown* subset, the use of expansion rules allows solve 23% of unaddressed instances and reduce the computation time in 15.4%. Regrettably, this success cannot be replicated in any `QF_LRA` logic subset.

With regards to the different types of rules expansion, only proportional projection and power regression rules allows to improve strategies efficiency. Thus, data is not modelable by others type of linear regression and/or the information used to relate the variables is insufficient implying a low correlation.

1. This sample has a size of 10 instances instead of 25 instances.

| Logic | Subset | Engine/Rules | Instances Set | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 25 instances | | | All instances | | |
| | | | solved | time[s] | total[s] | solved | time[s] | total[s] |
| LRA | *unknown* | Z3 | 22 | 775.56 | 7975.56 | 247 | 7324.12 | 91324.12 |
| | | *SequencEVO*$_{FS}$ | 22 | 27.17 | 7227.17 | 266 | 6951.99 | 45351.99 |
| | | **Proportion** | **23** | **489.34** | **5289.34** | **267** | **11361.79** | **47361.79** |
| | | Linear Reg. | 15 | 38.91 | 24038.91 | 216 | 3620.87 | 162020.87 |
| | | Exponential Reg. | 17 | 2346.99 | 21546.99 | 219 | 6943.16 | 158143.16 |
| | | Power Reg. | 23 | 2202.86 | 7002.86 | 264 | 19031.47 | 62231.47 |
| | | Quadratic Polynomial Reg. | 16 | 942.18 | 22542.18 | 215 | 2328.66 | 163128.66 |
| | | Fifth Polynomial Reg. | 15 | 80.25 | 24080.25 | 213 | 1188.56 | 166788.56 |
| QF_LIA | *known* | Z3 | **24** | **509.52** | **2909.52** | 5617 | 126486.39 | 659286.39 |
| | | *HybridEVO-1*$_{Z3}$ | 24 | 553.76 | 2953.76 | 5630 | 131832.62 | 633432.62 |
| | | **Proportion** | 24 | 2268.57 | 4668.57 | **5645** | **23465.04** | **618665.04** |
| | | Linear Reg. | 23 | 679.27 | 5479.27 | 5508 | 171311.76 | 1095311.76 |
| | | Exponential Reg. | 23 | 696.55 | 5496.55 | 5448 | 136744.97 | 1204744.97 |
| | | Power Reg. | 24 | 604.24 | 3004.24 | 5637 | 212016.26 | 826416.26 |
| | | Quadratic Polynomial Reg. | 24 | 1667.70 | 4067.70 | 5622 | 271631.68 | 922031.68 |
| | | Fifth Polynomial Reg. | 22 | 215.69 | 7415.69 | 5511 | 167898.61 | 1084698.61 |
| | *unknown* | Z3 | 10 | 37.46 | 36037.46 | 130 | 13230.82 | 426030.82 |
| | | *SequencEVO*$^{10}_{Z3}$ | 17 | 634.10 | 19834.10 | 214 | 21722.32 | 232922.32 |
| | | Proportion | 15 | 11.13 | 24011.13 | 215 | 15085.10 | 223885.10 |
| | | Linear Reg. | 17 | 506.79 | 19706.79 | 232 | 31275.28 | 199275.28 |
| | | Exponential Reg. | 17 | 517.08 | 19717.08 | 231 | 25903.12 | 196303.12 |
| | | **Power Reg.** | 17 | 499.22 | 19699.22 | **234** | **33958.77** | **197158.77** |
| | | Quadratic Polynomial Reg. | **17** | **308.27** | **19508.27** | 226 | 15728.95 | 198128.95 |
| | | Fifth Polynomial Reg. | 17 | 381.62 | 19581.62 | 220 | 9442.84 | 206242.84 |
| QF_LRA | *known* | Z3 | 23 | 1416.68 | 6216.68 | 1530 | 116198.57 | 346598.57 |
| | | ***StratEVO***$^{10}_{FS}$ | 24 | 278.14 | 2678.14 | **1583** | **44576.55** | **147845.73** |
| | | Proportion | 24 | 280.01 | 2680.01 | 1574 | 64031.96 | 188831.96 |
| | | Linear Reg. | 24 | 261.59 | 2661.59 | 1562 | 49710.43 | 203310.43 |
| | | Exponential Reg. | 24 | 258.62 | 2658.62 | 1565 | 54018.87 | 200418.87 |
| | | Power Reg. | 24 | 284.96 | 2684.96 | 1574 | 63347.34 | 188147.34 |
| | | Quadratic Polynomial Reg. | **24** | **168.58** | **2568.58** | 1528 | 30318.40 | 265518.40 |
| | | Fifth Polynomial Reg. | 24 | 280.14 | 2680.14 | 1551 | 42550.80 | 222550.80 |
| | *unknown*[1] | Z3 | 0 | 0.00 | 24000.00 | 2 | 2886.58 | 132486.58 |
| | | ***StratEVO***$_{FS}$ | 3 | 963.53 | 17763.53 | **38** | **1890.02** | **45090.02** |
| | | Proportion | 4 | 308.74 | 14708.74 | 10 | 9896.14 | 120296.14 |
| | | Linear Reg. | 6 | 4802.12 | 14402.12 | 33 | 31179.45 | 86379.45 |
| | | Exponential Reg. | **6** | **3831.27** | **13431.27** | 31 | 25175.13 | 85175.13 |
| | | Power Reg. | 6 | 7035.78 | 16635.78 | 31 | 37965.05 | 97965.05 |
| | | Quadratic Polynomial Reg. | 4 | 1079.84 | 15479.84 | 26 | 23559.23 | 95559.23 |
| | | Fifth Polynomial Reg. | 4 | 1551.30 | 15951.30 | 19 | 13518.63 | 102318.63 |

Table 9.2 – SMT-LIB Benchmarks: Expanding time configuration semantics of strategies generated by means of best StratEVO based engines.

| Logic | Subset | Engine/Rules | Instances Set | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 25 instances | | | All instances | | |
| | | | solved | time[s] | total[s] | solved | time[s] | total[s] |
| LRA | *unknown* | Z3 | 22 | 775.56 | 7975.56 | 247 | 7324.12 | 91324.12 |
| | | ***SequencEVO***$_{FS}$ | 22 | 27.17 | 7227.17 | **266** | **6951.99** | **45351.99** |
| | | Proportion | **23** | **991.92** | **5791.92** | 260 | 6914.78 | 59714.78 |
| | | Linear Reg. | 18 | 2619.40 | 19419.40 | 221 | 8666.31 | 155066.31 |
| | | Exponential Reg. | 23 | 1927.38 | 6727.38 | 266 | 26064.41 | 64464.41 |
| | | Power Reg. | 23 | 1792.30 | 6592.30 | 259 | 13452.95 | 68652.95 |
| | | Quadratic Polynomial Reg. | 23 | 2523.77 | 7323.77 | 262 | 18317.41 | 66317.41 |
| | | Fifth Polynomial Reg. | 18 | 2998.21 | 19798.21 | 220 | 7444.92 | 156244.92 |
| QF_LIA | *known* | Z3 | **24** | **509.52** | **2909.52** | 5617 | 126486.39 | 659286.39 |
| | | *HybridEVO-1*$_{Z3}$ | 24 | 553.76 | 2953.76 | 5630 | 131832.62 | 633432.62 |
| | | Proportion | 24 | 917.14 | 3317.14 | 5628 | 219010.56 | 855010.56 |
| | | Linear Reg. | 23 | 486.37 | 5286.37 | 5472 | 191810.62 | 1202210.62 |
| | | Exponential Reg. | 23 | 508.91 | 5308.91 | 5417 | 140565.11 | 1282965.11 |
| | | **Power Reg.** | 24 | 624.05 | 3024.05 | **5632** | **204151.44** | **830551.44** |
| | | Quadratic Polynomial Reg. | 24 | 1370.91 | 3770.91 | 5612 | 281428.58 | 955828.58 |
| | | Fifth Polynomial Reg. | 23 | 971.11 | 5771.11 | 5511 | 168366.78 | 1085166.78 |
| | *unknown* | Z3 | 10 | 37.46 | 36037.46 | 130 | 13230.82 | 426030.82 |
| | | *SequencEVO*$^{10}_{Z3}$ | 17 | 634.10 | 19834.10 | 214 | 21722.32 | 232922.32 |
| | | Proportion | 16 | 233.24 | 21833.24 | 214 | 4250.08 | 215450.08 |
| | | Linear Reg. | **17** | **346.84** | **19546.84** | 232 | 29402.82 | 197402.82 |
| | | **Exponential Reg.** | 17 | 524.76 | 19724.76 | **232** | **28355.58** | **196355.58** |
| | | Power Reg. | 15 | 9.72 | 24009.72 | 204 | 5204.16 | 240404.16 |
| | | Quadratic Polynomial Reg. | 16 | 137.90 | 21737.90 | 216 | 11590.54 | 217990.54 |
| | | Fifth Polynomial Reg. | 17 | 625.50 | 19825.50 | 232 | 29095.26 | 197095.26 |
| QF_LRA | *known* | Z3 | 23 | 1416.68 | 6216.68 | 1530 | 116198.57 | 346598.57 |
| | | ***StratEVO***$^{10}_{FS}$ | 24 | 278.14 | 2678.14 | **1583** | **44576.55** | **147845.73** |
| | | Proportion | 24 | 241.20 | 2641.20 | 1556 | 46578.85 | 214578.85 |
| | | Linear Reg. | **24** | **143.54** | **2543.54** | 1564 | 52218.49 | 201018.49 |
| | | Exponential Reg. | 24 | 276.15 | 2676.15 | 1566 | 55489.33 | 199489.33 |
| | | Power Reg. | 24 | 272.38 | 2672.38 | 1575 | 63826.95 | 186226.95 |
| | | Quadratic Polynomial Reg. | 24 | 276.08 | 2676.08 | 1575 | 63826.95 | 186226.95 |
| | | Fifth Polynomial Reg. | 24 | 283.58 | 2683.58 | 1576 | 66608.38 | 186608.38 |
| | *unknown*[1] | Z3 | 0 | 0.00 | 24000.00 | 2 | 2886.58 | 132486.58 |
| | | ***StratEVO***$_{FS}$ | 3 | 963.53 | 17763.53 | **38** | **1890.02** | **45090.02** |
| | | Proportion | 4 | 3340.28 | 17740.28 | 10 | 12417.28 | 122817.28 |
| | | Linear Reg. | 3 | 3700.78 | 20500.78 | 23 | 23698.23 | 102899.23 |
| | | Exponential Reg. | **7** | **6978.05** | **14178.05** | 33 | 30344.69 | 85544.69 |
| | | Power Reg. | 5 | 2183.37 | 14183.37 | 10 | 6014.63 | 116414.63 |
| | | Quadratic Polynomial Reg. | 0 | 0.00 | 0.00 | 1 | 2151.41 | 134151.41 |
| | | Fifth Polynomial Reg. | 0 | 0.00 | 0.00 | 1 | 2163.37 | 134163.37 |

Table 9.3 – SMT-LIB Benchmarks: Expanding time configuration semantics of semantically equivalent strategies generated by means of best StratEVO based engines.

#### 9.2.1.1.2 Adapting semantically equivalent strategies

The use of semantically equivalent strategies also allows to improve strategies performance in some SMT logics. As Table 9.3 summarises, expand time semantics in these strategies allows to slightly improve instance solving in `QF_LIA` logic set, despite not triggering same performance as the original strategies. In `QF_LIA` *known*, expanded strategies correctly address around 1% of unsolved instances, with a bigger effort of 31% more computation time. In its *unknown* counterpart, the use of the expanded strategy allows to solve 21% of unaddressed instances using 16% less execution time, being this last scenario very similar to the use of time expanded original strategies. Pitifully, these kind of expanded strategies have more struggles with SMT logics. Therefore, it cannot improve performance in LRA *unknown* and `QF_LRA` logic subsets.

As well as time expanded original strategies, power regression based rules allows to improve strategies efficiency. Also, exponential regression enhance solver performance. The remaining types of linear regression, and thus the dependant variable information, are insufficient to improve Z3 performance.

### 9.2.2 Expanded Strategies: Original vs Semantically Equivalent

In order to evaluate how different are the original expanded strategies with regards to the semantically equivalent strategies, we evaluate their performance by using the different expanding rules. Table 9.4 summarises a set of Two-Tailed Student T Test for means of paired samples, with significance level of $\alpha = 0.05$. They measure the average performance of the strategies generated in each logic after being modified by the different expansion rules. Thus, we compare the average performance depending on the initial strategy, that is to say, the original or semantically equivalent expanded strategy. These tests stand statistically significant difference between the performance of both alternatives using instance solving outputs.

In the mentioned Table, we could check that, with exception of `QF_LIA` *known* subset, there is no significance difference between the output of generated by both type of expanded strategies. Therefore, the equivalence rules applied to the original strategy do not change its semantics.

| Logic | Subset | Type | Diff. Mean | RMSD | $t$ value | $p$ value | better | status |
|---|---|---|---|---|---|---|---|---|
| `LRA` | *unknown* | instances | 15.67 | 3093.33 | 1.54 | **0.183512** | Equivalent | ✗ |
| `QF_LIA` | *known* | instances | -16.5 | 1037.50 | -2.81 | **0.037736** | Original | ✓ |
| | *unknown* | instances | -4.67 | 1015.33 | -0.80 | **0.458870** | Original | ✗ |
| `QF_LRA` | *known* | instances | 9.17 | 2386.83 | 1.03 | **0.351221** | Equivalent | ✗ |
| | *unknown* | instances | -11.83 | 654.83 | -2.54 | **0.052350** | Original | ✗ |

Table 9.4 – Student T Test: Statistical significance between expanded strategies, with level $\alpha = 0.05$.

## 9.3 Conclusions

In this Chapter, we extended the set of rules used to generate strategies outside evolutionary process of the selected engines. We included rules that allows to map the time semantics of the original generated strategies into other execution environment, specially for the SMT-COMP scenario. Obtained results imply that mapping time configuration semantics allows to improve Z3 solver performance in some SMT logic, e.g. `QF_LIA` logic where we outperformed all existent configurations.

Also, we propose the use of semantically equivalent strategies as alternative. The time semantics of these strategies could be easily expanded, thus the focus is on the components whose performance depends on the strategy time configuration. Despite not having better results as the original expansion, these are capable of outperforms some of best known results, as `QF_LIA` logic set.

However, the use of simple expanding and regression models, bounds the potential of these approaches in some SMT logics, e.g. `QF_LRA` logic. Hence, the use of multivariated regression analysis can lead to more successful performances and better semantic expansion. Moreover, these expansion procedure could be focused in other strategies components, as combinators or others parameter configurations.

# 10

# Conclusions and Perspectives

In this chapter we summarise and conclude over contributions of this thesis. Also, we discuss and define steps to follow for future works and related fields.

## 10.1 Research Contributions

In this thesis, we introduced a framework for the automated generation of strategies for SMT solvers. This procedure relies on a set of rules to modify and generate new strategies. The rule system acts as an intermediate layer, between well-known algorithms and strategies. Thus, our scheme allows to apply different engines without the need of adapting or modifying them structurally.

Through the empirical use of this framework, we shown that it suits as an autonomous search tool for evaluating and selecting different algorithms in order to generate optimised strategies by means of rules applications.

The main contributions of this research are:

— **A framework for automated generation of strategies**: We build a framework that defines several components and stages necessaries to generate optimised strategies. These elements includes core components, learning components, and execution components. Core components includes a set of rules and an initial strategy. Rules act as an intermediate layer between algorithms and the initial strategies, and define how algorithms will modify and generate new strategies. This intermediate layer also allows apply different type of algorithms without need of modify their structure or create complex versions of

them. Learning components includes a learning sample, i.e., set of instances, to train and generate strategies, a learning time-out per instance, and a global learning budget. We highlight the used of reduced learning samples in order to generate efficient strategies as alternative for logics whose benchmarks are hard to address, i.e., numerous or complex, reducing computational resources. Execution components allow to validate the models and results obtained in the learning process.

— **Address the Strategy Challenge in SMT [10] with a autonomous procedure**: Through the application of our framework in an autonomous environment, i.e., an empirical algorithm selection process for strategy optimisation in Z3, we shown that end-users could exert strategic control over core heuristic aspects of high-performance SMT solvers without need of expert knowledge. Note that the framework implementation was achieved using several evolutionary approaches, modification rules, and SMT logics. Specifically, modification rules, the core of the framework, were designed according to the syntax of Z3 strategies. This design did not include in-depth analysis of strategies components effects.

— **Address the "*Search for strategies rather than instances*" challenge [2] in Search-based Software Engineering**: Improving SMT solvers performance through generating optimised strategies, builds more reliable systems. This have a direct impact on the trend of mapping software engineering problems as SMT instances, i.e., search problems, because more classes of Search-based Software Engineering (SBSE) problems could be addressed with a single technique. Therefore, our framework allow to improve the resolution of several classes of SBSE problem, by reducing it to a single class of SBSE problem: *Software Improvement*.

— **Introduce SMT [4] as useful systematic search system for Combinatorial Optimisation**: The design and use of a hybrid approach that includes SMT solvers and meta-heuristics have been successfully to improve Z3 performance. Solved SMT instances and benchmarks include several SBSE problems which are commonly expressed as search problems, i.e., combinatorial optimisation problems. This fact demonstrates that SMT tools could address well-known and classic Combinatorial Optimisation problems, i.e., search problems, which are generally view as outside the scope of SMT development.

## 10.2 Future Work

Building upon this research, there are several projection of this work to the future. We discuss some of them below.

— Include more semantic-based rules for the strategy generation process. While, we demonstrated that some semantics rules for time selection improved generated strategy effi-

ciency, there are other several sources to infer this kind of rules, including: valid syntax construction with no solving impact and/or empirical inference from generated strategies.

— Apply strategies generation framework in SMT instance concerning another first-order logics related to software development, as bit-vector, array and pseudo-boolean logics. These logics covers several software engineering and computer science topics, including several problems which could be mapped as search instances.

— Address the construction of strategies for complex SMT logics by using the information of those that composes them. Several SMT problems include logics which are interrelated, e.g. `QF_ALIA` includes theory of arrays and linear integer arithmetics, use the information of their components as engine starting points should be useful for generate more robust strategies.

— Propagate rule selection framework to be used with other metaheuristics procedures and different class of problems. The context of the Strategy Challenge in SMT [10] of generating a strategies driven by a well-known language, helped to easily apply evolutionary approaches as Genetic Programming. However, other several metaheuristic tools could be used as engine of this framework, e.g. bio-inspired algorithms or single-solution search techniques. Moreover, this framework aims to be applied in several different context, and should be proven in several combinatorial optimisation problems.

— Finally, use and evaluate SMT as systematic and complete tools for solve well-known Combinatorial Optimisation Problems. Actually, classic complete search techniques as SAT solving and Constraint Programming procedures have been widely used to address optimisation problems, specially in constraint satisfaction problems. SMT should be considered an efficient alternative to address these problem in a standalone environment, but also in hybrid systems where complete search tools interact and cooperate with incomplete and local search tools.

## 10.3 Scientific Publications

The scientific contributions achieved during this thesis has been constantly published in diverse scientific journals, international conferences and workshops. Our goal is to spread our research in order to expand the efforts done in the field. The novel way as our framework and engines are exposed, the contributions achieved and the promising results obtained show that there is still several work to be done and there are many possibilities to innovate and create new knowledge. To date, our list of scientific publications is the following:

— Journal Articles:

1. **Improving complex SMT strategies with learning** by Nicolás Gálvez Ramírez, Eric Monfroy, Frédéric Saubion and Carlos Castro. In *International Transaction in Operational Research*. IFORS. Wiley. 2018. (submitted on June 15th, 2018) [181].

— International Conferences:

1. **Optimizing SMT Solving Strategies by Learning with an Evolutionary Process** by Nicolás Gálvez Ramírez, Eric Monfroy, Frédéric Saubion and Carlos Castro. In *International Conference on High Performance Computing & Simulation : PACOS 2018*. Orléans, France. 2018 [182].
2. **Evolving SMT Strategies** by Nicolás Gálvez Ramírez, Youssef Hamadi, Eric Monfroy and Frédéric Saubion. In *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*. San Jose, CA, USA. 2016 [183].
3. **Towards Automated Strategies in Satisfiability Modulo Theory** by Nicolás Gálvez Ramírez, Youssef Hamadi, Eric Monfroy and Frédéric Saubion. In *Genetic Programming. EuroGP 2016*. Porto, Portugal. 2016 [184].

— International Workshops:

1. **Generation and Optimization of SMT Strategies** by Nicolás Gálvez Ramírez, Eric Monfroy, Frédéric Saubion and Carlos Castro. In *Workshop on Optimization and Learning: Challenges and Applications - OLA*. Alicante, Spain. 2018 [185].

# Bibliography


[1] M. Grötschel and L. Lovász, "Combinatorial Optimization: A Survey," tech. rep., DIMACS, 1993.

[2] M. Harman, "The role of Artificial Intelligence in Software Engineering," in *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012 First International Workshop on*, pp. 1–6, June 2012.

[3] W. Visser, N. Bjørner, and N. Shankar, "Software Engineering and Automated Deduction," in *Proceedings of the on Future of Software Engineering*, FOSE 2014, pp. 155–166, ACM, 2014.

[4] L. De Moura and N. Bjørner, "Satisfiability Modulo Theories: Introduction and Applications," *Commun. ACM*, vol. 54, pp. 69–77, Sept. 2011.

[5] S. A. Cook, "The Complexity of Theorem-proving Procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pp. 151–158, ACM, 1971.

[6] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008.*, LNCS, pp. 337–340, Springer, 2008.

[7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pp. 171–177, Springer-Verlag, 2011.

[8] B. Dutertre, "Yices 2.2," in *Computer Aided Verification* (A. Biere and R. Bloem, eds.), pp. 737–744, Springer International Publishing, 2014.

[9] L. de Moura and N. Bjørner, "Applications and Challenges in Satisfiability Modulo Theories," in *Second International Workshop on Invariant Generation, WING 2009, York, UK, March 29, 2009 and Third International Workshop on Invariant Generation, WING 2010, Edinburgh, UK, July 21, 2010*, pp. 1–11, 2010.

[10] L. de Moura and G. Passmore, "The Strategy Challenge in SMT Solving," in *Automated Reasoning and Mathematics*, LNCS, pp. 15–44, Springer, 2013.

[11] D. E. Knuth, "A Terminological Proposal," *SIGACT News*, vol. 6, pp. 12–18, Jan. 1974.

[12] D. E. Knuth, "Postscript About NP-hard Problems," *SIGACT News*, vol. 6, pp. 15–16, Apr. 1974.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[14] M. Sipser, *Introduction to the Theory of Computation*. International Thomson Publishing, 1st ed., 1996.

[15] C. M. Papadimitriou, *Computational complexity*. Addison-Wesley, 1994.

[16] M. Gendreau and J.-Y. Potvin, *Handbook of Metaheuristics*. Springer Publishing Company, Incorporated, 2nd ed., 2010.

[17] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd ed., 2015.

[18] J. Bell and B. Stevens, "A survey of known results and research areas for n-queens," *Discrete Mathematics*, vol. 309, no. 1, pp. 1 – 31, 2009.

[19] H. Hoos and T. Sttzle, *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., 2004.

[20] A. Sangiovanni-Vincentelli, "Editor's foreword," *Algorithmica*, vol. 6, pp. 295–301, Jun 1991.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *SCIENCE*, vol. 220, no. 4598, pp. 671–680, 1983.

[22] F. Glover, "Tabu Search—Part I," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[23] T. A. Feo, M. G. C. Resende, and S. H. Smith, "A Greedy Randomized Adaptive Search Procedure for Maximum Independent Set," *Oper. Res.*, vol. 42, pp. 860–878, Oct. 1994.

[24] A. W. Johnson, *Generalized hill climbing algorithms for discrete optimization problems*. PhD thesis, VirginiaTech, Oct. 1996.

[25] H. R. Lourenço, O. C. Martin, and T. Stützle, *Iterated Local Search*, pp. 320–353. Springer US, 2003.

[26] A. Blot, M.-É. Kessaci, and L. Jourdan, "Survey and unification of local search techniques in metaheuristics for multi-objective combinatorial optimisation," *Journal of Heuristics*, May 2018.

[27] L. Paquete, M. Chiarandini, and T. Stützle, "Pareto local optimum sets in the biobjective traveling salesman problem: An experimental study," in *Metaheuristics for Multiobjective Optimisation* (X. Gandibleux, M. Sevaux, K. Sörensen, and V. T'kindt, eds.), pp. 177–199, Springer Berlin Heidelberg, 2004.

[28] M. M. Drugan and D. Thierens, "Stochastic pareto local search: Pareto neighbourhood exploration and perturbation strategies," *Journal of Heuristics*, vol. 18, pp. 727–766, Oct 2012.

[29] E. Angel, E. Bampis, and L. Gourvès, "Approximating the pareto curve with local search for the bicriteria tsp(1,2) problem," *Theoretical Computer Science*, vol. 310, no. 1, pp. 135 – 146, 2004.

[30] C. Darwin, *On the origin of species by means of natural selection, or, The preservation of favoured races in the struggle for life*. John Murray, 1859.

[31] T. Bäck, *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Inc., 1996.

[32] L. Fogel, A. Owens, and M. Walsh, *Artificial intelligence through simulated evolution*. Wiley, 1966.

[33] L. J. Fogel, A. J. Owens, and M. J. Walsh, "Intelligent decision making through a simulation of evolution," *Behavioral Science*, vol. 11, no. 4, pp. 253–272.

[34] L. J. Fogel, *Intelligence Through Simulated Evolution: Forty Years of Evolutionary Programming*. John Wiley & Sons, Inc., 1999.

[35] I. Rechenberg, *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Problemata, Frommann-Holzboog, 1973.

[36] H.-P. Schwefel, *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*. ISR, Birkhaeuser, 1977.

[37] J. Holland, "Genetic Algorithms and the Optimal Allocation of Trials," *SIAM Journal on Computing*, vol. 2, no. 2, pp. 88–105, 1973.

[38] J. H. Holland, *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[39] K. A. De Jong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, 1975.

[40] J. R. Koza, "Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems," tech. rep., 1990.

[41] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[42] J. R. Koza, D. Andre, F. H. Bennett, and M. A. Keane, *Genetic Programming III: Darwinian Invention & Problem Solving*. Morgan Kaufmann Publishers Inc., 1st ed., 1999.

[43] J. R. Koza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[44] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'Neill, "Grammar-based Genetic Programming: a survey," *Genetic Programming and Evolvable Machines*, vol. 11, pp. 365–396, Sep 2010.

[45] P. A. Whigham, "Grammatically-based Genetic Programming," in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pp. 33–41, 9 1995.

[46] C. Ryan, J. J. Collins, and M. O'Neill, "Grammatical Evolution: Evolving Programs for an Arbitrary Language," in *Proceedings of the First European Workshop on Genetic Programming*, EuroGP '98, pp. 83–96, Springer-Verlag, 1998.

[47] M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[48] L. Vanneschi, M. Castelli, and S. Silva, "A survey of semantic methods in genetic programming," *Genetic Programming and Evolvable Machines*, vol. 15, pp. 195–214, Jun 2014.

[49] A. Moraglio, K. Krawiec, and C. G. Johnson, "Geometric Semantic Genetic Programming," in *Parallel Problem Solving from Nature - PPSN XII* (C. A. C. Coello, V. Cutello, K. Deb, S. Forrest, G. Nicosia, and M. Pavone, eds.), pp. 21–31, Springer Berlin Heidelberg, 2012.

[50] Q. Nguyen and D. S. o. C. S. . I. University College, *Examining Semantic Diversity and Semantic Locality of Operators in Genetic Programming*. PhD thesis, University College Dublin, 2011.

[51] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, pp. 127–145, Jun 1968.

[52] T. S. Hussain and R. A. Browse, "Attribute Grammars for Genetic Representations of Neural Networks and Syntactic Constraints of Genetic Programming," in *Proceedings of the Workshop on Evolutionary Computation*, Jun 1998.

[53] M. de la Cruz Echeandía, A. O. de la Puente, and M. Alfonseca, "Attribute Grammar Evolution," in *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach* (J. Mira and J. R. Álvarez, eds.), pp. 182–191, Springer Berlin Heidelberg, 2005.

[54] H. Christiansen, "A Survey of Adaptable Grammars," *SIGPLAN Not.*, vol. 25, pp. 35–44, Nov. 1990.

[55] A. Ortega, M. de la Cruz, and M. Alfonseca, "Christiansen Grammar Evolution: Grammatical Evolution With Semantics," *IEEE Transactions on Evolutionary Computation*, vol. 11, pp. 77–90, Feb 2007.

[56] F. C. Pereira and D. H. Warren, "Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks," *Artificial Intelligence*, vol. 13, no. 3, pp. 231 – 278, 1980.

[57] M. L. Wong and K. S. Leung, "An induction system that learns programs in different programming languages using genetic programming and logic grammars," in *Proceedings of 7th IEEE International Conference on Tools with Artificial Intelligence*, pp. 380–387, Nov 1995.

[58] M. L. Wong and K. S. Leung, "Combining genetic programming and inductive logic programming using logic grammars," in *Proceedings of 1995 IEEE International Conference on Evolutionary Computation*, vol. 2, pp. 733–736 vol.2, Nov 1995.

[59] L. Beadle and C. G. Johnson, "Semantically driven crossover in genetic programming," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pp. 111–116, June 2008.

[60] L. Beadle and C. G. Johnson, "Semantically driven mutation in genetic programming," in *2009 IEEE Congress on Evolutionary Computation*, pp. 1336–1342, May 2009.

[61] K. Krawiec and T. Pawlak, "Locally geometric semantic crossover: a study on the roles of semantics and homology in recombination operators," *Genetic Programming and Evolvable Machines*, vol. 14, pp. 31–63, Mar 2013.

[62] K. Krawiec and T. Pawlak, "Approximating Geometric Crossover by Semantic Backpropagation," in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pp. 941–948, ACM, 2013.

[63] L. Vanneschi, *An Introduction to Geometric Semantic Genetic Programming*, pp. 3–42. Springer International Publishing, 2017.

[64] N. F. McPhee, B. Ohs, and T. Hutchison, "Semantic Building Blocks in Genetic Programming," in *Genetic Programming* (M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcázar, I. De Falco, A. Della Cioppa, and E. Tarantino, eds.), pp. 134–145, Springer Berlin Heidelberg, 2008.

[65] E. Burke, S. Gustafson, and G. Kendall, "A survey and analysis of diversity measures in genetic programming," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, GECCO'02, pp. 716–723, Morgan Kaufmann Publishers Inc., 2002.

[66] N. F. McPhee and N. J. Hopper, "Analysis of Genetic Diversity Through Population History," in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*, GECCO'99, pp. 1112–1120, Morgan Kaufmann Publishers Inc., 1999.

[67] E. K. Burke, S. Gustafson, and G. Kendall, "Diversity in genetic programming: an analysis of measures and correlation with fitness," *IEEE Transactions on Evolutionary Computation*, vol. 8, pp. 47–62, Feb 2004.

[68] L. Beadle and C. G. Johnson, "Semantic Analysis of Program Initialisation in Genetic Programming," *Genetic Programming and Evolvable Machines*, vol. 10, pp. 307–337, Sept. 2009.

[69] M. Looks, "On the Behavioral Diversity of Random Programs," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pp. 1636–1642, ACM, 2007.

[70] J. P. Bowen and M. G. Hinchey, eds., *Applications of Formal Methods*. Prentice Hall PTR, 1st ed., 1995.

[71] C. G. Johnson, "Deriving Genetic Programming Fitness Properties by Static Analysis," in *Genetic Programming* (J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. Tettamanzi, eds.), pp. 298–307, Springer Berlin Heidelberg, 2002.

[72] C. G. Johnson, "Genetic Programming with Guaranteed Constraints," in *Applications and Science in Soft Computing* (A. Lotfi and J. M. Garibaldi, eds.), pp. 95–100, Springer Berlin Heidelberg, 2004.

[73] C. G. Johnson, "Genetic Programming with Fitness Based on Model Checking," in *Genetic Programming* (M. Ebner, A. O'Neill, Michaeland Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, eds.), pp. 114–124, Springer Berlin Heidelberg, 2007.

[74] G. Katz and D. Peled, "Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms," in *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, ATVA '08, pp. 33–47, Springer-Verlag, 2008.

[75] J. R. Rice, "The Algorithm Selection Problem," vol. 15 of *Advances in Computers*, pp. 65 – 118, Elsevier, 1976.

[76] Y. Hamadi, E. Monfroy, and F. Saubion, *Autonomous Search*. Springer Publishing Company, Incorporated, 2012.

[77] Y. Hamadi, E. Monfroy, and F. Saubion, *An Introduction to Autonomous Search*, pp. 1–11. Springer Berlin Heidelberg, 2012.

[78] A. E. Eiben and S. K. Smit, *Evolutionary Algorithm Parameters and Methods to Tune Them*. Springer Berlin Heidelberg, 2012.

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

[80] A. E. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter control in evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 2, pp. 124–141, 1999.

[81] F. G. Lobo, C. F. Lima, and Z. Michalewicz, *Parameter Setting in Evolutionary Algorithms*. Springer Publishing Company, Incorporated, 1st ed., 2007.

[82] E. Montero, M.-C. Riff, and B. Neveu, "A beginner's guide to tuning methods," *Applied Soft Computing*, vol. 17, pp. 39 – 51, 2014.

[83] A. Eiben and S. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 19 – 31, 2011.

[84] M. Birattari, *Tuning Metaheuristics: A Machine Learning Perspective*. Springer Publishing Company, Incorporated, 1st ed. 2005. 2nd printing ed., 2009.

[85] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A Racing Algorithm for Configuring Metaheuristics," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, GECCO'02, pp. 11–18, Morgan Kaufmann Publishers Inc., 2002.

[86] P. Balaprakash, M. Birattari, and T. Stützle, "Improvement Strategies for the F-Race Algorithm: Sampling Design and Iterative Refinement," in *Hybrid Metaheuristics* (T. Bartz-Beielstein, M. J. Blesa Aguilera, C. Blum, B. Naujoks, A. Roli, G. Rudolph, and M. Sampels, eds.), pp. 108–122, Springer Berlin Heidelberg, 2007.

[87] T. W. MacFarland and J. M. Yates, *Friedman Twoway Analysis of Variance (ANOVA) by Ranks*, pp. 213–247. Springer International Publishing, 2016.

[88] F. Hutter, H. H. Hoos, and T. Stützle, "Automatic Algorithm Configuration Based on Local Search," in *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pp. 1152–1157, AAAI Press, 2007.

[89] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An Automatic Algorithm Configuration Framework," *J. Artif. Int. Res.*, pp. 267–306, 2009.

[90] B. Adenso-Diaz and M. Laguna, "Fine-Tuning of Algorithms Using Fractional Experimental Designs and Local Search," *Oper. Res.*, vol. 54, pp. 99–114, Jan. 2006.

[91] S. K. Smit and A. E. Eiben, "Beating the world champion evolutionary algorithm via REVAC tuning," in *IEEE Congress on Evolutionary Computation*, pp. 1–8, July 2010.

[92] M. Riff and E. Montero, "A new algorithm for reducing metaheuristic design effort," in *2013 IEEE Congress on Evolutionary Computation*, pp. 3283–3290, June 2013.

[93] A. Aleti and I. Moser, "A Systematic Literature Review of Adaptive Parameter Control Methods for Evolutionary Algorithms," *ACM Comput. Surv.*, vol. 49, pp. 56:1–56:35, Oct. 2016.

[94] R. Battiti, M. Brunato, and F. Mascia, *Reactive Search and Intelligent Optimization*. Springer Publishing Company, Incorporated, 1st ed., 2008.

[95] R. Battiti and M. Brunato, *Reactive Search Optimization: Learning While Optimizing*, pp. 543–571. Springer US, 2010.

[96] R. Battiti and P. Campigotto, *An Investigation of Reinforcement Learning for Reactive Search Optimization*, pp. 131–160. Springer Berlin Heidelberg, 2012.

[97] R. Battiti and G. Tecchiolli, "The Reactive Tabu Search," *ORSA Journal on Computing*, vol. 6, no. 2, pp. 126–140, 1994.

[98] F. Mascia, P. Pellegrini, M. Birattari, and T. Stützle, "An analysis of parameter adaptation in reactive tabu search," *International Transactions in Operational Research*, vol. 21, no. 1, pp. 127–152.

[99] S. Kawaguchi and Y. Fukuyama, "Reactive Tabu Search for Job-shop scheduling problems considering peak shift of electric power energy consumption," in *2016 IEEE Region 10 Conference (TENCON)*, pp. 3406–3409, Nov 2016.

[100] L. Ingber, "Adaptive simulated annealing (ASA): Lessons learned," *Control and Cybernetics*, vol. 25, pp. 33–54, 1996.

[101] H. Aguiar e Oliveira Junior, L. Ingber, A. Petraglia, M. Rembold Petraglia, and M. Augusta Soares Machado, *Adaptive Simulated Annealing*, pp. 33–62. Springer Berlin Heidelberg, 2012.

[102] L. F. Fraga-Gonzalez, R. Q. Fuentes-Aguilar, A. García-González, and G. Sánchez-Ante, "Adaptive simulated annealing for tuning PID controllers," *AI Commun.*, vol. 30, pp. 347–362, 2017.

[103] M. Brunato and R. Battiti, *RASH: A Self-adaptive Random Search Method*, pp. 95–117. Springer Berlin Heidelberg, 2008.

[104] R. Battiti and P. Campigotto, *An Investigation of Reinforcement Learning for Reactive Search Optimization*, pp. 131–160. Springer Berlin Heidelberg, 2012.

[105] G. Karafotias, M. Hoogendoorn, and A. E. Eiben, "Parameter Control in Evolutionary Algorithms: Trends and Challenges," *IEEE Transactions on Evolutionary Computation*, vol. 19, pp. 167–187, April 2015.

[106] F. G. Lobo and C. F. Lima, "A Review of Adaptive Population Sizing Schemes in Genetic Algorithms," in *Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation*, GECCO '05, pp. 228–234, ACM, 2005.

[107] F. G. Lobo and C. F. Lima, *Adaptive Population Sizing Schemes in Genetic Algorithms*, pp. 185–204. Springer Publishing Company, Incorporated, 1st ed., 2007.

[108] Á. Fialho, *Adaptive Operator Selection for Optimization*. PhD thesis, Université Paris Sud - Paris XI, Dec. 2010.

[109] J. Maturana and F. Saubion, "A Compass to Guide Genetic Algorithms," in *Parallel Problem Solving from Nature - PPSN X, 10th International Conference Dortmund, Germany, September 13-17, 2008, Proceedings*, pp. 256–265, 2008.

[110] J. Maturana, Á. Fialho, F. Saubion, M. Schoenauer, F. Lardeux, and M. Sebag, *Adaptive Operator Selection and Management in Evolutionary Algorithms*, pp. 161–189. Springer Berlin Heidelberg, 2012.

[111] G. di Tollo, F. Lardeux, J. Maturana, and F. Saubion, "An experimental study of adaptive control for evolutionary algorithms," *Applied Soft Computing*, vol. 35, pp. 359 – 372, 2015.

[112] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833 – 839, 2001.

[113] Harman, M., "The Relationship Between Search Based Software Engineering and Predictive Modeling," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pp. 1:1–1:13, ACM, 2010.

[114] T. Menzies and G. Koru, "Predictive models in software engineering," *Empirical Software Engineering*, vol. 18, pp. 433–434, Jun 2013.

[115] M. Shtern and V. Tzerpos, "Clustering Methodologies for Software Engineering," *Adv. Soft. Eng.*, Jan. 2012.

[116] D. Zhang and J. J. P. Tsai, "Machine learning and software engineering," in *14th IEEE International Conference on Tools with Artificial Intelligence, 2002. (ICTAI 2002). Proceedings.*, pp. 22–29, Nov 2002.

[117] D. J. M. S. (auth.), *Automated Theorem Proving in Software Engineering*. Springer-Verlag Berlin Heidelberg, 1 ed., 2001.

[118] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based Software Engineering: Trends, Techniques and Applications," *ACM Comput. Surv.*, pp. 11:1–11:61, Dec. 2012.

[119] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, *Search Based Software Engineering: Techniques, Taxonomy, Tutorial*, pp. 1–59. Springer Berlin Heidelberg, 2012.

[120] P. McMinn, "Search-Based Software Testing: Past, Present and Future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 153–163, March 2011.

[121] J. A. Clark, H. Dan, and R. M. Hierons, "Semantic mutation testing," *Science of Computer Programming*, vol. 78, no. 4, pp. 345 – 363, 2013.

[122] F. Ferrucci, M. Harman, and F. Sarro, *Search-Based Software Project Management*, pp. 373–399. Springer Berlin Heidelberg, 2014.

[123] M. O'Keeffe and M. O. Cinneide, "Search-based software maintenance," in *Conference on Software Maintenance and Reengineering (CSMR'06)*, March 2006.

[124] M. O'Keeffe and M. O. Cinneide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502 – 516, 2008.

[125] O. Räihä, "A survey on search-based software design," *Computer Science Review*, vol. 4, no. 4, pp. 203 – 249, 2010.

[126] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang, "Search Based Software Engineering for Software Product Line Engineering: A Survey and Directions for Future Work," in *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, (New York, NY, USA), pp. 5–18, ACM, 2014.

[127] A. Zeller, "Search-Based Program Analysis," in *Search Based Software Engineering* (M. B. Cohen and M. Ó Cinnéide, eds.), pp. 1–4, Springer Berlin Heidelberg, 2011.

[128] W. B. Langdon and M. Harman, "Optimizing Existing Software With Genetic Programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, pp. 118–135, Feb 2015.

[129] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic Improvement of Software: A Comprehensive Survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, pp. 415–432, June 2018.

[130] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated Software Transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pp. 257–269, ACM, 2015.

[131] M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," *Journal of Systems and Software*, vol. 1, pp. 213 – 221, 1979.

[132] M. Harman, "Software Engineering Meets Evolutionary Computation," *Computer*, vol. 44, pp. 31–39, Oct 2011.

[133] S. Malik and L. Zhang, "Boolean Satisfiability from Theoretical Hardness to Practical Success," *Commun. ACM*, vol. 52, pp. 76–82, Aug. 2009.

[134] J. Franco and J. Martin, "A History of Satisfiability," in *Handbook of Satisfiability*, pp. 3–74, 2009.

[135] M. R. Prasad, A. Biere, and A. Gupta, "A Survey of Recent Advances in SAT-based Formal Verification," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, pp. 156–173, Apr. 2005.

[136] S. Chaki, "SAT-Based Software Certification," in *Tools and Algorithms for the Construction and Analysis of Systems* (H. Hermanns and J. Palsberg, eds.), pp. 151–166, Springer Berlin Heidelberg, 2006.

[137] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based bounded model checking for software verification," *Theoretical Computer Science*, vol. 404, no. 3, pp. 256 – 274, 2008. International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004).

[138] J. P. Galeotti, N. Rosner, C. G. L. Pombo, and M. F. Frias, "TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1283–1307, Sept 2013.

[139] F. Arito, F. Chicano, and E. Alba, "On the Application of SAT Solvers to the Test Suite Minimization Problem," in *Search Based Software Engineering* (G. Fraser and J. Teixeira de Souza, eds.), pp. 45–59, Springer Berlin Heidelberg, 2012.

[140] N. Bjørner, K. L. McMillan, and A. Rybalchenko, "Program Verification as Satisfiability Modulo Theories," in *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, pp. 3–11, 2012.

[141] N. Bjørner, K. L. McMillan, and A. Rybalchenko, "Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Data-types," *CoRR*, vol. abs/1306.5264, 2013.

[142] R. M. Karp, *Reducibility among Combinatorial Problems*, pp. 85–103. Springer US, 1972.

[143] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[144] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, pp. 201–215, July 1960.

[145] J. P. M. Silva and K. A. Sakallah, "Grasp-a new search algorithm for satisfiability," in *Proceedings of International Conference on Computer Aided Design*, pp. 220–227, Nov 1996.

[146] J. P. Marques-Silva and K. A. Sakallah, "Grasp: a search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, pp. 506–521, May 1999.

[147] R. J. Bayardo, Jr. and R. C. Schrag, "Using csp look-back techniques to solve real-world sat instances," in *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pp. 203–208, AAAI Press, 1997.

[148] D. Monniaux, "A Survey of Satisfiability Modulo Theory," in *Computer Algebra in Scientific Computing* (V. P. Gerdt, W. Koepf, W. M. Seiler, and E. V. Vorozhtsov, eds.), pp. 401–425, Springer International Publishing, 2016.

[149] K. R. M. Leino, "Automating Theorem Proving with SMT," in *Interactive Theorem Proving* (S. Blazy, C. Paulin-Mohring, and D. Pichardie, eds.), pp. 2–16, Springer Berlin Heidelberg, 2013.

[150] T. Weber, "SMT solvers: new oracles for the HOL theorem prover," *International Journal on Software Tools for Technology Transfer*, vol. 13, pp. 419–429, Oct 2011.

[151] S. Böhme, *Proving Theorems of Higher-Order Logic with SMT Solvers*. PhD thesis, Technische Universität München, 2012.

[152] M. Bofill, J. Coll, J. Suy, and M. Villaret, "Solving the Multi-Mode Resource-Constrained Project Scheduling Problem with SMT," in *2016 IEEE 28th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 239–246, Nov 2016.

[153] Z. Cheng, H. Zhang, Y. Tan, and Y. Lim, "SMT-based scheduling for multiprocessor real-time systems," in *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pp. 1–7, June 2016.

[154] J. Rintanen, "Discretization of Temporal Models with Application to Planning with SMT," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pp. 3349–3355, AAAI Press, 2015.

[155] J. Rintanen, "Temporal Planning with Clock-based SMT Encodings," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pp. 743–749, AAAI Press, 2017.

[156] D. Bryce, S. Gao, D. Musliner, and R. Goldman, "SMT-based Nonlinear PDDL+ Planning," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, AAAI Press, 2015.

[157] G. Nelson and D. C. Oppen, "Simplification by Cooperating Decision Procedures," *ACM Trans. Program. Lang. Syst.*, vol. 1, Oct. 1979.

[158] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)." `www.SMT-LIB.org`, 2018.

[159] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," tech. rep., Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[160] SMT-LIB Community, "SMT-LIB Logics." Avialable at http://smtlib.cs.uiowa.edu/logics.shtml.

[161] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump, "6 years of SMT-COMP," *Journal of Automated Reasoning*, pp. 243–277, 2013.

[162] M. Heinzmann, A. Niemetz, G. Regers, and T. Weber, "11th International Satisfiability Modulo Theories Competition (SMT-COMP 2018): Rules and Procedures." Available in http://smtcomp.sourceforge.net/2017/rules17.pdf.

[163] M. Heinzmann, G. Regers, and T. Weber, "12th International Satisfiability Modulo Theories Competition (SMT-COMP 2017): Rules and Procedures." Available in http://smtcomp.sourceforge.net/2017/rules17.pdf.

[164] S. Conchon, D. Déharbe, M. Heinzmann, and T. Weber, "11th International Satisfiability Modulo Theories Competition (SMT-COMP 2016): Rules and Procedures." Available in http://smtcomp.sourceforge.net/2016/rules16.pdf.

[165] S. Conchon, D. Déharbe, and T. Weber, "10th International Satisfiability Modulo Theories Competition (SMT-COMP 2015): Rules and Procedures." available in http://smtcomp.sourceforge.net/2015/rules15.pdf.

[166] D. R. Cok, D. Déharbe, and T. Weber, "The 2014 SMT competition," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, pp. 207–242, 2014.

[167] C. Barrett, L. de Moura, and A. Stump, "Design and results of the 1st Satisfiability Modulo Theories Competition (SMT-COMP 2005)," *Journal of Automated Reasoning*, vol. 35, no. 4, pp. 373–390, 2005.

[168] G. Sutcliffe, "The CADE ATP System Competition - CASC," *AI Magazine*, vol. 37, no. 2, pp. 99–101, 2016.

[169] T. Balyo, M. J. H. Heule, and M. Järvisalo, "SAT competition 2016: Recent developments," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pp. 5061–5063, 2017.

[170] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pp. 93–107, Springer-Verlag, 2013.

[171] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The OpenSMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (J. Esparza and R. Majumdar, eds.), pp. 150–153, Springer Berlin Heidelberg, 2010.

[172] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "The Barcelogic SMT Solver," in *Computer Aided Verification* (A. Gupta and S. Malik, eds.), pp. 294–298, Springer Berlin Heidelberg, 2008.

[173] S. Graham-Lengrand, "Slot Machines: an approach to the Strategy Challenge in SMT solving," in *13th International Workshop on Satisfiability Modulo Theories*, July 2015.

[174] S. Graham-Lengrand, "Psyche: A Proof-Search Engine Based on Sequent Calculus with an LCF-Style Architecture," in *Automated Reasoning with Analytic Tableaux and Related Methods: 22nd International Conference, TABLEAUX 2013*, pp. 149–156, Springer, 2013.

[175] Longman, *The Longman Dictionary of Contemporary English*. Pearson Longman, 6th edition ed., 2014.

[176] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[177] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward, *A Classification of Hyper-heuristic Approaches*, pp. 449–468. Springer US, 2010.

[178] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi, "Tree Automata Techniques and Applications." Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[179] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.

[180] M. Research, "Z3 Online Documentation." Available on: https://github.com/Z3Prover/z3/wiki/Documentation.

[181] N. Gálvez Ramírez, E. Monfroy, F. Saubion, and C. Castro, "Improving complex SMT strategies with learning." Submitted in June 15th, 2018.

[182] N. Gálvez Ramírez, E. Monfroy, F. Saubion, and C. Castro, "Optimizing SMT Solving Strategies by Learning with an Evolutionary Process," in *International Conference on High Performance Computing & Simulation : Pacos 2018*, 2018.

[183] 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)*, pp. 247–254, Nov 2016.

[184] N. Gálvez Ramírez, Y. Hamadi, E. Monfroy, and F. Saubion, "Towards Automated Strategies in Satisfiability Modulo Theory," in *Genetic Programming: 19th European Conference, EuroGP 2016*, pp. 230–245, Springer, 2016.

[185] N. Gálvez Ramírez, E. Monfroy, F. Saubion, and C. Castro, "Generation and optimization of smt strategies," in *Workshop on Optimization and Learning: Challenges and Applications - OLA*, (Alicante), 2018.

**Titre :** Un Cadre pour la Génération Autonome de Stratégies dans la Satisfiabilité Modulo des Théories.

**Mots clés :** Satisfiabilité Modulo des Théories, Algorithmes Évolutionnaires, Métaheuristiques, Recherche Autonome, Réglage de Paramètres.

**Résumé :** La génération de stratégies pour les solveurs en Satisfiabilité Modulo des Théories (SMT) nécessite des outils théoriques et pratiques qui permettent aux utilisateurs d'exercer un contrôle stratégique sur les aspects heuristiques fondamentaux des solveurs de SMT, tout en garantissant leur performance. Nous nous intéressons dans cette thèse au solveur Z3, l'un des plus efficaces lors des compétitions SMT (SMT-COMP). Dans les solveurs SMT, la définition d'une stratégie repose sur un ensemble de composants et paramètres pouvant être agencés et configurés afin de guider la recherche d'une preuve de (in)satisfiabilité d'une instance donnée.
Dans cette thése, nous abordons ce défi en définissant un cadre pour la génération autonome de stratégies pour Z3, c'est-à-dire un algorithme qui permet de construire automatiquement des stratégies sans faire appel à des connaissances d'expertes. Ce cadre général utilise une approche évolutionnaire (programmation génétique), incluant un système à base de règles. Ces règles formalisent la modification de stratégies par des principes de réécriture, les algorithmes évolutionnaires servant de moteur pour les appliquer. Cette couche intermédiaire permettra d'appliquer n'importe quel algorithme ou opérateur sans qu'il soit nécessaire de modifier sa structure, afin d'introduire de nouvelles informations sur les stratégies. Des expérimentations sont menées sur les jeux classiques de la compétition SMT-COMP.

**Title :** A Framework for Autonomous Generation of Strategies in Satisfiability Modulo Theories

**Keywords :** Satisfiability Modulo Theories, Evolutionary Algorithms, Metaheuristics, Autonomous Search, Parameter Configuration

**Abstract :** The Strategy Challenge in Satisfiability Modulo Theories (SMT) claims to build theoretical and practical tools allowing users to exert strategic control over core heuristic aspects of high performance SMT solvers. In this work, we focus in Z3 Theorem Prover: one of the most eficient SMT solver according to the SMT Competition, SMT-COMP. In SMT solvers, the definition of a strategy relies on a set of tools that can be scheduled and configured in order to guide the search for a (un)satisfiability proof of a given instance.
In this thesis, we address the Strategy Challenge in SMT defining a framework for the autonomous generation of strategies in Z3, i.e., a practical system to automatically generate SMT strategies without the use of expert knowledge. This framework is applied through an incremental evolutionary approach starting from basic algorithms to more complex genetic constructions. This framework formalise strategies modification as rewriting rules, where algorithms acts as engines to apply them. This intermediate layer, will allow apply any algorithm or operator with no need to being structurally modified, in order to introduce new information in strategies. Validation is done through experiments on classic benchmarks of the SMT-COMP.