RESOLVER

BOOLEAN SATISFIABILITY SOLVER BASED ON A GENETIC ALGORITHM

Software Requirements Specification

Team Imperium

Dewald de Jager
Ernst Eksteen
Vignesh Iyer
Regan Koopmans
Craig van Heerden

October 18, 2017

Contents

1 Introduction

1.1 Purpose

Requirements engineering is a complex task that has a high impact on the time-to-market, cost and overall success of a project. The purpose of this software requirements specification is to lay out the findings of a standardized requirements elicitation for the proposed application. In doing this, both the capabilities that the application is expected to deliver together, with the constraints on the solution space are described.

The document will take into account requirements, as set out by both the developers and the prospective users, in order to lay a foundation for the up-coming phases of the Software Development Cycle. The various sections each provide details on specific types of requirements of the application.

The intended audience include developers/peers(Team Imperium), domain experts(Prof. Stefan Gruner and Dr Nils Timm), customers/end-users(researchers at various institutions) who will perform a technical, expert and customer review of this document respectively.

1.2 Scope

The proposed software (Resolver) serves to act as a research tool/utility that implements a hybrid algorithm for the satisfiability problem. The hybrid approach combines local search and genetic search to determine whether a given input propositional logic formula is satisfiable or not. The system is intended to improve on and replace the current SAT solver that is being used at the University of Pretoria.

The benefits of GASAT are that specific crossover operators are used to identify particularly promising search areas, while local search performs an intensified search of solutions around these areas. In such a way, a good compromise between intensification and diversification in the search procedure is achieved. The definition of specific crossover operators take into account the semantic aspects of the SAT problem. The algorithm avoids brute-force techniques of exploring the entire search space as were employed by predecessor algorithms. The objective is for this algorithm to considerably enlarge our capacity of solving large and hard SAT instances.

1.3 Requirements Engineering Methodology

It is very important to follow a tried-and-tested methodology as a guide for what should be documented and how, and to avoid making common mistakes. In this specification we use the requirements engineering methodology as laid out in *Requirements Engineering: Fundamentals, Principles, and Techniques* [2]. This methodology complements our Agile software engineering methodology as it promotes continuous requirements engineering. There are many benefits to treating requirements engineering tasks as cross-lifecycle activities, but the most beneficial to this project are that the requirements stay up to date and are used systematically. The responsibilities of the system and its components are made very clear.

The Pohl requirements engineering methodology identifies three type of requirement artifacts:

1. Goal-Oriented Requirements
2. Scenario-Oriented Requirements
3. Solution-Oriented Requirements

Goal-oriented requirements are specified at a very high level and state what the system is supposed to achieve. Scenario-oriented requirements describe concrete examples of satisfying or failing to satisfy goals; They are meant to illustrate the goals with more detail and are typically defined through use of user stories and use-case diagrams. Finally, solution-oriented requirements define the data, functions, behaviour, quality and constraints of the system. These requirements have been divided further into three solution-oriented requirement artifacts:

1. Functional Requirements: Define the services that should be provided by the system

2. Data Dictionary: Defines the data types of data flowing through the system
3. Behavioural Requirements: Defines algorithmic information that cannot be captured by functional requirements, such as execution-order

These artifacts are the focus of the remainder of this document and are explained in more detail in the relevant sections.

1.4 Definitions, Acronyms and Abbreviations

For this paper, in acquiesce with other literature, we will use the abbreviation *SAT* to refer to the concept of the *boolean satisfiability problem.*

Definitions:
Propositional Logic : A sub-field of logic that deals with the evaluation and manipulation of true and false statements.
Genetic Algorithm: An algorithm that uses concepts from biological evolution to optimize a certain property within a certain constrained environment.
Local Search:
Crossover: A crossover is the critical reproductive operator used to determine how a novel solution is derived from one or more parents
Atom: A single variable or literal in a boolean formula.
Assignment:

Acronyms and Abbreviations:
CDFD: Condition Data Flow Diagram
CNF: Conjunctive Normal Form
GASAT: Genetic Local Search Algorithm for the Satisfiability Problem
RE: Requirements Engineering
SAT: Boolean satisfiability
SDLC: Software Development Life Cycle
SOFL: Structured Object-Oriented Formal Language

2 Overall Description

2.1 Product Perspective

Explain the product at a very-high level. Explain the main features of the product. Resolver is an open-source implementation of the GASAT algorithm [1].

2.2 User Characteristics

This software is targeted at a very specific, niche need of academics and practitioners in this field. We therefore do not expect this to be a mainstream product, in the sense that it would not be used by the average individual. Consequently, we make the assumption that an end-user of the system has the following characteristics:

- Has average computer literacy.
- Has a reasonable expectation of what they want to achieve with proposed software.
- Has an understanding of Propositional Logic.
- Has prior knowledge of Evolutionary Computation (specifically genetic algorithms) in order to be able to configure the parameters used by this algorithm.
- Has a good understanding of the boolean satisfiability problem.

2.2.1 User Class 1 - The Researcher

The single most important use is that of the university researchers for which the software is commissioned. For this reason, we believe the following functions are essential for this user class.

- Ability to modify solver parameters, and compare/benchmark these in terms of performance, an generally accommodate experimentation in the nature of solving.
- Be precise, accurate and dependable in program results to such a high degree, such that program outcomes may be used research material.

2.2.2 User Class 2 - The Problem Solver

The software can also be used by an individual who wishes to solve a specific instances of variable substitution, rather than to explore field as and end to itself. This person might be validating the correctness of an algorithm, proving a mathematical theorem, or any problem that can be represented with sufficient abstraction. This brings about an entirely different set of functional requirements, which may summarized as follows:

- Ability to accommodate a large class of real-world problems.
- Providing logical defaults and predefined templates.

2.2.3 User Class 3 - The Learner

The software can be used as a learning tool for educating students and curious individuals on the nature of boolean satisfiability. To this user class, we ascribe the following requirements:

- Have an accessible user interface, that is intuitive and embodies the Principle of Least Astonishment (POLA).
- Provide helpful information that supports the use the software to a novice, or someone who is otherwise unfamiliar with the particular details of the algorithms used.

2.3 Assumptions and Dependencies

For this project we assume that that the formula provided to the software has been pre-encoded. In other words, our software need not be concerned with what these logic statements represent, and how they are derived from a specific application domain.

The project will depend on the runtime environments specific to the technologies. For instance, we will use Python for processing and Electron for the interface.

3 Requirements

3.1 Goal-Oriented

TODO: Mention goal decomposition (Slide 55) and dependencies and then refactor this section

- G1 The system should accept a propositional logic formula as input in DIMACS format.
    - G1.1 The system should accept input in the form of an external file.
    - G1.2 The system should enable input via direct entry through the means of some form in the user-interface.
    - G1.3 The system should accept input programmatically through a predefined API.
- G2 The system should support the capacity for multiple user-interfaces.
    - G2.1 The system should allow for a purely text-based control and display mechanism.
    - G2.2 The system should support a widely compatible graphical desktop application.
    - G2.3 The system should allow for a web browser-based interface.

G3 The system should work independently from an external user-interface.

G4 The system should implement the GASAT algorithm correctly.

G5 The system should enable customization of the genetic algorithm.

- G5.1 The parameters of the genetic algorithm should be customizable.
- G5.2 The cross-over operator of the genetic algorithm should be selectable.

G6 The tabu search should improve search performance by avoiding local optima.

G7 The system should support customization of tabu search.

- G7.1 The parameters of the tabu search should be customizable.
- G7.2 The tabu search should conditionally offer a diversification modifier.
- G7.3 The tabu search should conditionally offer a RCVF modifier.

G8 The genetic algorithm should create and maintain an initially random population of candidate solutions.

G9 The genetic algorithm should utilize a fitness measure, as defined in [1], based on the number of satisfied clauses in the formula.

G10 The genetic algorithm should implement a selection operator to the end of cross-over.

- G10.1 The genetic algorithm should emit a sub-population based on Elitism.
- G10.2 The genetic algorithm should select two parents from the sub-population using Tournament selection.

G11 The genetic algorithm should generate new candidate solutions by applying the cross-over to two parent candidate solutions.

G12 The genetic algorithm should only ever replace the worst individual with a new candidate solution if the new candidate solution is at least as fit as the wost individual in the sub-population.

G13 The genetic algorithm must iterate so long as it has not found a solution, or it has exceeded the maximum allocated generations.

G14 The system should output the result of an attempt to satisfy the propositional logic formula.

- G14.1 The system should report a solution, if one is found.
- G14.2 The system should report the best candidate solution, if no satisfying solution is found.
- G14.3 The system should report the unsatisfied clauses if no satisfying solution is found.

G15 The system should gather statistics during execution so that they can be reported on completion

- G15.1 The system should record the number of generations that have passed
- G15.2 The system should record the execution time
- G15.3 The system should record the best solution and its fitness value as it progresses

G16 The system should allow spontaneous termination of the solving process.

3.2 Scenario-Oriented

3.2.1 User Stories

- As a Researcher, it is important that the software is dependable and robust, such that I may use it in my research.
- As a Researcher, I want sufficient control over the details of the solving process, in the aim of producing novel experimental results on boolean satisfiability.
- As a Student, I want the software to be simple to learn and support my understanding of boolean satisfiability.
- As a Student, it is important to me that the software omits details of SAT solving that I may not yet understand or be concerned with.
- As a Problem Solver, it is important to me that the software is efficient and effective, and performs competitively in comparison to other contemporary SAT solvers.
- As a Problem Solver, I want the software to accommodate a significant portion of SAT problems, and choose reasonable defaults for parameters that I may not be concerned with optimizing.
- As a general user, I want the software to be easy to start and interact with.

3.3 Solution-Oriented

3.3.1 Functional Requirements

A semi-formal specification of the processes and data stores in the system can be found in this section. The functional requirements "define services the system should provide, behaviour of the system and in some cases also what the system should not do." [2]

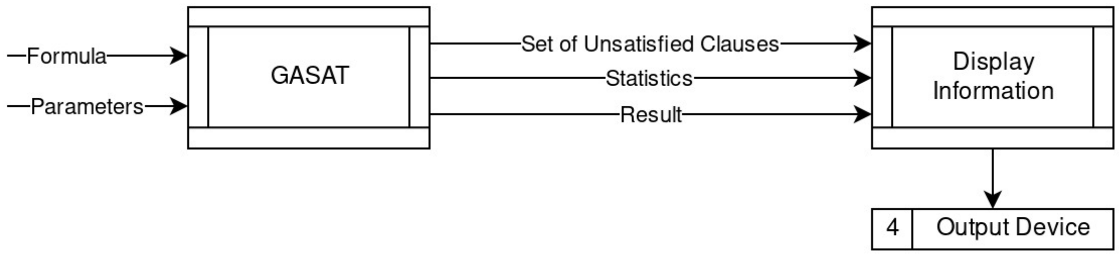

Figure 1: High-level overview of the core algorithm

We start by describing the `GASAT` process seen in Figure 1. The `GASAT` process takes two inputs, namely the propositional logic formula to be satisfied and the parameters for configuring the genetic algorithm and tabu search. The input `Parameters` has a data type of `ParametersType` which is a composite type we formally define. This allows us to group all the parameters and their appropriate data types together. The `crossover_operator` field in the `ParametersType` type is declared as being of type `CrossoverOperator`. This is an enumeration type we formally define and can take on the values `CC` for the Corrective Clause crossover, `CCTM` for the Corrective Clause with Truth Maintenance crossover and `FF` for the Fleurent and Ferland crossover operator.

The `GASAT` process has 3 outputs, namely the set of unsatisfied clauses, the statistics and the result. The unsatisfied clauses is a set of integers that reference the clauses in the formula that could not be satisfied if a satisfying solution could not be found. The result is the satisfying solution that was found, if any. When the set of unsatisfied clauses is the empty set, the result must contain a solution and when the result contains no solution, the set of unsatisfied clauses may not be empty. The `Statistics` output is of a composite type called `StatisticsType` that we also define.

The preconditions of the `GASAT` process specify constraints on the inputs and the process will only be activated when all inputs are present and the preconditions are true. An additional precondition is placed on the `Formula` specifying that it must be in DIMACS format. The postconditions specify constraints on the output and should evaluate to true if the process has executed correctly.

The `Display Information` process takes the output of the `GASAT` process as input and therefore has preconditions similar to the `GASAT` process's postconditions. The `Display Information` process also has write access to an external data store called `Output Device`. The postcondition only ensures that something was written to the `Output Device` data store as it may be the satisfying result, partial result, statistics, set of unsatisfied clauses, an error message or any combination of these.

```
CrossoverOperator = {<CC>, <CCTM>, <FF>}

ParametersType = composed of
max_generations: int
    population_size: int
    sub_population_size: int
    crossover_operator: CrossoverOperator
    tabu_list_length: int
    maximum_flips: int
    rvcf: bool
    diversification: bool
    max_false: int
    recursion_count: int
    flip_constraint: int
end

StatisticsType = composed of
percentage_solved: real
    generations_passed: int
end

process GASAT(Formula: string, Parameters: ParametersType) Set_Of_Unsatisfied_Clauses:
              set of int, Statistics: StatisticsType, Result: string
pre
len(Formula) > 12
    and Parameters.max_generations > 0
    and Parameters.population_size >= 2
    and Parameters.sub_population_size >= 2
    and Parameters.sub_population_size <= Parameters.population_size
    and Parameters.tabu_list_length > 0
    and Parameters.maximum_flips > 0
    and Parameters.max_false > 0
    and Parameters.recursion_count > 0
    and Parameters.flip_constraint > 0
post
    (card(Set_Of_Unsatisfied_Clauses) > 0 and len(Result) = 0
    or card(Set_Of_Unsatisfied_Clauses) = 0 and len(Result) > 0)
    and Statistics.percentage_solved >= 0
    and Statistics.percentage_solved <= 100
    and Statistics.generations_passed > 0
    and Statistics.generations_passed <= Parameters.max_generations
end_process

process Display_Information(Set_Of_Unsatisfied_Clauses: set of int, Statistics:
                            StatisticsType, Result: string)
ext wr Output_Device: string
pre
(card(Set_Of_Unsatisfied_Clauses) > 0 and len(Result) = 0
    or card(Set_Of_Unsatisfied_Clauses) = 0 and len(Result) > 0)
    and Statistics.percentage_solved >= 0
    and Statistics.percentage_solved <= 100
    and Statistics.generations_passed > 0
    and Statistics.generations_passed <= Parameters.max_generations
post
len(Output_Device) > len(~Output_Device)
end_process
```

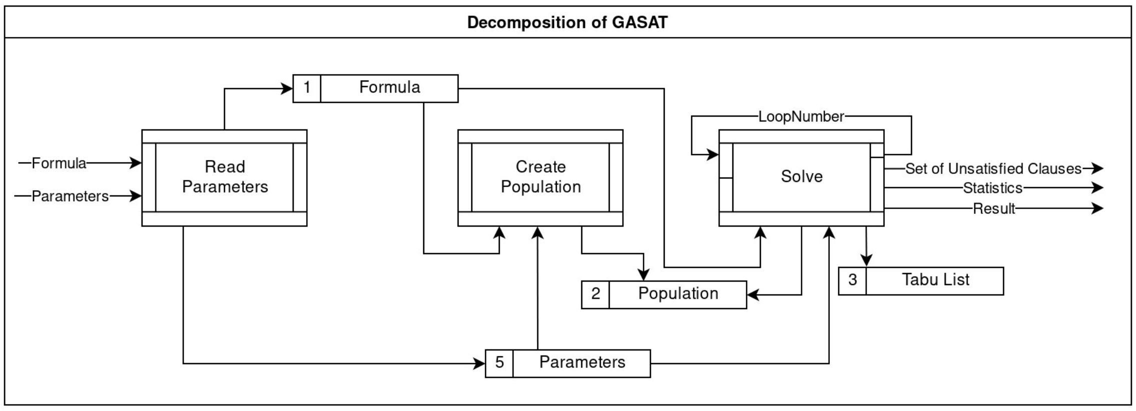

Figure 2: Decomposition of the GASAT process in the core algorithm

The GASAT process depicted in Figure 1 is decomposed in Figure 2. The first process in the decomposition, Read Parameters, receives the same input as the GASAT process and therefore has the same preconditions. This process parses the input formula and writes it to the Formula data store as a set of clauses which are, in turn, sequences of integers. Additionally, the parameters that are read in are persisted in the Parameters data store. The only precondition ensures that the Formula data store is non-empty which indicates that the input formula was successfully parsed.

The Create Population process gains control once the Read Parameters process has completed successfully, otherwise the program terminates with an error message. The Create Population process generates a random population and ensures that the population is of the correct size. This process reads from the Formula data store to evaluate quality of generated solutions and reads the size of the population to be generated from the Parameters data store. The generated population is written to the Population data store.

Once the Create Population process has terminated successfully, control is given to the Solve process. This process represents the iterative procedure of the algorithm. The only input is a LoopNumber which should initially be 0 and will increment on completion of every iteration until either a satisfying solution is found or the maximum number of iterations (generations) has passed. The output of this process is similar to that of the GASAT process but with an additional case for when more iterations are necessary, in which only an updated LoopNumber is outputted. This process reads the Formula and Parameters data stores and writes to both the Population and Tabu List data stores.

```
process Read_Parameters(Formula: string, Parameters: ParametersType)
ext wr Formula: set of seq of int
pre
len(Formula) > 12
    and Parameters.max_generations > 0
    and Parameters.population_size >= 2
    and Parameters.sub_population_size >= 2
    and Parameters.sub_population_size <= Parameters.population_size
    and Parameters.tabu_list_length > 0
    and Parameters.maximum_flips > 0
    and Parameters.max_false > 0
    and Parameters.recursion_count > 0
    and Parameters.flip_constraint > 0
post
    card(Formula) > 0
end_process

process Create_Population()
ext rd Formula: set of seq of int
ext rd Parameters: ParametersType
ext wr Population: set of string
pre
    card(Formula) > 0
post
card(Population) = Parameters.population_size
end_process

process Solve(LoopNumber: int) Set_Of_Unsatisfied_Clauses: set of int, Statistics: StatisticsType,
              Result: string
ext rd Formula: set of seq of int
ext rd Parameters: ParametersType
ext wr Population: set of string
ext wr Tabu_List: set of int
pre
LoopNumber = 0
    and card(Formula) > 0
    and card(Population) = Parameters.population_size
post
    (card(Set_Of_Unsatisfied_Clauses) > 0 and len(Result) = 0
    or card(Set_Of_Unsatisfied_Clauses) = 0 and len(Result) > 0)
    and Statistics.percentage_solved >= 0
    and Statistics.percentage_solved <= 100
    and Statistics.generations_passed = LoopNumber
    and Statistics.generations_passed > 0
    and Statistics.generations_passed <= Parameters.max_generations
end_process
```

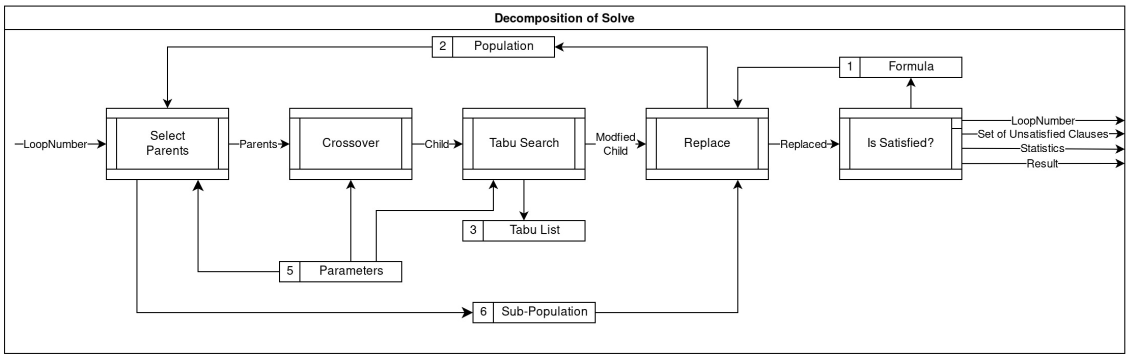

Figure 3: Decomposition of the Solve process within the GASAT process

The `Solve` process depicted in Figure 2 is decomposed in Figure 3. The first process, `Select Parents`, receives the same input as the `Solve` process. It verifies that the `LoopNumber` is in a valid range and that the population has been generated. It then selects a subset of the `Population` data store to write to the `Sub-Population` data store. The desired size of the sub-population is read from the `Parameters` data store. Two candidate solutions are then read from the sub-population to form the output set, `Parents`.

The `Crossover` process activates once parents have been selected by the selection operator and produces a new candidate solution, `Child`. The crossover operator that should be used to create this child is read from the `Parameters` data store. `Child` is then used as input to the `Tabu`

`Search` process which reads and modifies the `Tabu List` data store. This tabu list is used as a small memory that aids in avoiding local minima. The `Child` may or may not be modified by `Tabu Search` and is then passed on as input to the `Replace` process as the `ModifiedChild`. The `Replace` process compared the fitness, or quality, of the new candidate solution and replaces the worst solution in the `Population` data store only if the new candidate solution has a fitness value lower than the worst solution in the `Sub-Population` data store. If the new candidate solution was placed in the population the output `Replaced` is true, and false otherwise.

The process `Is Satisfied?` evaluates the candidate solution to see whether it satisfies the formula. If the candidate solution is a satisfying solution or the maximum number of generations has passed, the iteration stops and the result, set of unsatisfied clauses and statistics are used as output. If the above condition does not hold, the `LoopNumber` is incremented and is used as output instead, resulting in another iteration.

```
process Select_Parents(LoopNumber: int) Parents: set of string
ext rd Parameters: ParametersType
ext rd Population: set of string
ext wr Sub_Population: set of string
pre
    LoopNumber >= 0
    and LoopNumber < Parameters.max_generations
    and card(Population) = Parameters.population_size
post
    card(Sub_Population) = Parameters.sub_population_size
    and len(Parents) = 2
end_process

process Crossover(Parents: set of string) Child: string
ext rd Parameters: ParametersType
pre
    len(Parents) = 2
post true
end_process

process Tabu_Search(Child: string) ModifiedChild: string
ext rd Parameters: ParametersType
ext wr Tabu_List: set of int
pre true
post
    card(Tabu_List) >= 0
    and card(Tabu_List) <= len(Child)
end_process

process Replace(ModifiedChild: string) Replaced: bool
ext rd Sub_Population: set of string
ext rd Formula: set of seq of int
ext wr Population: set of string
pre true
post
    card(Population) = Parameters.population_size
end_process

process Is_Satisfied(Replaced: bool) Set_Of_Unsatisfied_Clauses: set of int, Statistics:
                     StatisticsType, Result: string | LoopNumber: int
ext rd Sub_Population: set of string
ext rd Formula: set of seq of int
ext wr Population: set of string
pre true
post
    (card(Set_Of_Unsatisfied_Clauses) > 0 and len(Result) = 0
    or card(Set_Of_Unsatisfied_Clauses) = 0 and len(Result) > 0)
    and Statistics.percentage_solved >= 0
    and Statistics.percentage_solved <= 100
    and LoopNumber = ~LoopNumber + 1
    and Statistics.generations_passed = LoopNumber
    and Statistics.generations_passed > 0
    and Statistics.generations_passed <= Parameters.max_generations
end_process
```

3.3.2 Data Dictionary

3.3.3 Behavioural Requirements

- The main entry-point of the system is the GASAT process.
- The Create Population process may not start until the Read Parameters process has completed.
- The Solve process may only terminate once a solution has been found or the maximum number of generations has passed.
- *Correctness* - The SAT Solver must fulfill the internal mathematical consistency properties by application of the SOFL method for design.
- *Reliability* - The functioning of the SAT solver should not deteriorate as the input formula becomes more complex.
- *Usability* - The user interface needs to be both simple to navigate about, while at the same time exposing all necessary use-cases of the SAT Solver.
- *Integration* - The SAT Solver must be an easily pluggable tool within the larger context of a software verification program.
- *Portability* - The SAT Solver must be executable on multiple hardware and software platforms.
- *Performance* - Variable substitution problems are inherently slow. We therefore need to strive to make the system as performance as possible, such that it allows for efficient use by a researcher.
- *Modifiability* - The various internal control parameters and algorithms should be easily editable as and when the need arises to tweak them.

4 Prioritization and Release Plan

We have chosen to use an Agile project management method, and will therefore base our development around feature-driven goals and bi-weekly meetings with the client.

4.1 Choice of Prioritization Method

We will be prioritizing the textual operation of the GASAT program, since this is the use case that is most important to our client as a researcher in this field. After this feature, we will prioritize the user interface of the program,

A GASAT Author Specification

The GASAT algorithm as presented in [1] is decomposed by the author and specified using pseudocode and mathematical functions. These specifications are extracted from the paper and presented here for reference.

References

[1] Frédéric Lardeux, Frédéric Saubion, and Jin-Kao Hao. Gasat: a genetic local search algorithm for the satisfiability problem. *Evolutionary Computation*, 14(2):223–253, 2006.

[2] Klaus Pohl. *Requirements engineering: fundamentals, principles, and techniques.* Springer Publishing Company, Incorporated, 2010.