TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software and Multimedia Technology

Chair of Software Technology

Systematic Parameter Tuning of Genetic Optimization Approaches

Roman Kosovnenko

roman.kosovnenko@mailbox.tu-dresden.de
Born on: 18th January 1994 in Chernihiv
Course: Distributed Systems Engineering
Matriculation number: 4733290
Matriculation year: 2017

Master Thesis

to achieve the academic degree

Master of Science (M.Sc.)

Supervisors
M.Sc. Dmytro Pukhkaiev
Dipl.-Inf. Johannes Mey
Dr.-Ing. Sebastian Götz

Supervising professor
Prof. Dr. rer. nat habil. Uwe Aßmann

Submitted on: 9th March 2020

Acknowledgment

Thanks for ALL

Contents

1 Introduction 7
1.1 Motivation . . . 7
1.2 Objective . . . 7
1.3 Overview . . . 7

2 Background 8
2.1 Multi Quality Auto Tunning (MQuAT) . . . 8
2.1.1 Design principles of MQuAT . . . 8
2.1.2 Operation principles of MQuAT . . . 8
2.1.3 MQuAT combine the design time and runtime . . . 9
2.2 MQuAT problem . . . 10
2.3 The solution of the MQuAT problem . . . 12
2.4 Genetic algorithm . . . 13
2.4.1 Selector . . . 13
2.4.2 crossover . . . 14
2.4.3 mutation . . . 14
2.5 Genetic solver . . . 15
2.5.1 Tree Shape Genotype . . . 15
2.5.2 Crossover operator . . . 15
2.5.3 Mutation operator . . . 17
2.6 Parameter Tuning Strategies for GA . . . 17
2.7 BRISE . . . 17

3 The way to get better results 19
3.1 Parameter tuning as a beginning of live . . . 19
3.2 What if we could change magic numbers or Explain the code to ducks 21
3.3 Astrologers have announced a week of probabilities. The number of probabilities doubled. . . . 22
3.4 Peppa Pig and adaptation for own life . . . 23
3.5 Unique genotypes is a God sign or placebo for genetic algorithms . . . 26

4 Evaluation and Analysis 27
4.1 Evaluation . . . 27
4.1.1 Benchmark set . . . 27
4.1.2 Benchmark results . . . 27
4.1.3 Compare with random solver . . . 27
4.2 Analysis . . . 27

5 Conclusion 28

6 Future work 29

Bibliography 30

List of Figures

2.1 Three layers of MQuAT 9
2.2 Combined structure of MQuAT 10
2.3 Hardware meta-model 11
2.4 Software meta-model 11
2.5 MQuAT problem model 12
2.6 Main loop of the genetic algorithm 13
2.7 Example of the crossover 14
2.8 Example of the mutation 14
2.9 Example of the Tree Shape Genotype 16
2.10 Crossover in Tree Shape Genotype 16
2.11 Mutation in Tree Shape Genotype 17
2.12 The high-level architecture of BRISE 17

3.1 Boxplot with a number of contract violations for the basic version of genetic solver 20
3.2 Boxplot with a number of contract violations for the basic version of genetic solver and with tuned parameters 21
3.3 Boxplot with a number of contract violations for the genetic solver without hardcoded parameters in comparison with previous versions 22
3.4 Boxplot with a number of contract violations for the genetic solver with added probabilities without tuning in comparison with previous versions 23
3.5 Boxplot with a number of contract violations for the genetic solver with added probabilities and tuned parameters in comparison with previous versions 24
3.6 Boxplot with a number of contract violations for the genetic solver with adaptive crossover rate in comparison with previous versions 25
3.7 Boxplot with a number of contract violations for the genetic solver without duplicates in the population in comparison with previous versions 26

List of Tables

1 Introduction

Many optimization problems could be solved using evolutionary programming (e.g. Genetic Algorithm).

1.1 Motivation

-

1.2 Objective

1.3 Overview

The description of this thesis is organized as follows: In Chapter 2, we extend not advanced in field of problem solving and optimizations reader by background knowledge and defines scope of thesis. Chapter 3 describes related work in defined scope. In Chapter 4 one will find the concept description of dynamic heuristics selection. Chapter 5 contains more detailed information about approach implementation and embedding it to BRISE. The evaluation re-

2 Background

This thesis based on several kinds of research of SAS, auto-tuning, evolutionary algorithms, and software optimization. This chapter summarizes the important aspects and details of approaches and optimization technics used in the succeeding chapters.

2.1 Multi Quality Auto Tunning (MQuAT)

Multi-Quality Auto-Tuning (MQuAT) – is an approach to self-adaptive software, which provides design and operation principles for software systems that automatically provide the best possible utility to the user while producing the least possible cost [3]. It is based on the design-time part, which represents a new development method for self-optimizing systems and runtime parts, which concerns operation principles, namely, novel techniques to runtime self-optimization [3].

2.1.1 Design principles of MQuAT

MQuAT presented a new method of developing self-optimized software. In which software is proposed to be constructed from components with specifically defined limits. In addition, components are intended to comprise multiple implementations, each providing the same but differing functionality in their non-functional behavior. Therefore, the design principle is the critical factor for runtime optimization because when there is a different configuration, optimization can be done, and the optimal or almost optimal configuration can be selected. To compare different implementations of the software component, they need to be specified with their non-functional properties (NFP) [3]. A new meta-architecture of the self-optimizing software system was created To highlight this fundamental principle. It called the cool component model [6]. A specialty of MQuAT is the application of QoS contracts to cover non-functional implementation behavior as well as the interrelationships of different components between NFPs. Contracts naturally describe the relationship between provisions and requirements.

2.1.2 Operation principles of MQuAT

The core runtime approach proposed in MQuAT is the THE Auto-Tuning Runtime Environment (THEATRE)[4, 6]. The concept of runtime environment contains three layers: a user, software, and a resource layer. All of them depicted in Figure 2.1

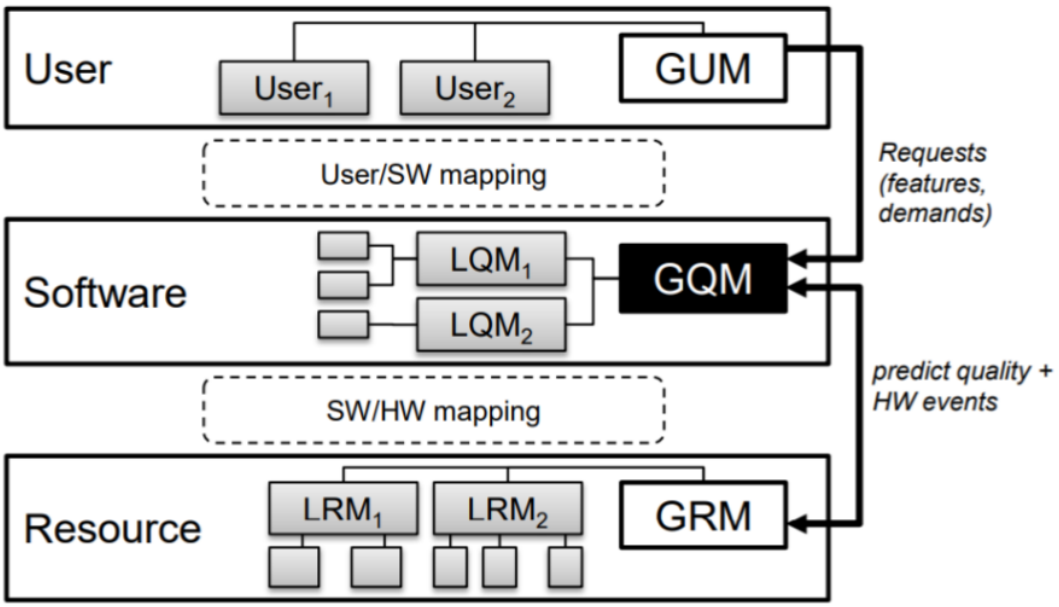

Figure 2.1 Three layers of MQuAT

The user layer invokes features and identifies their specifications. The Global User Manager (GUM) is required to manage the mapping between users and software component implementations and to coordinate the requests. User requests are minimal software system specifications that have to be met to satisfy the user. The general aim of the runtime system is to help the user as effectively as possible about the purposes of the user [3].

The software layer contains all software components, and it's implementations. Each Component has a Local Quality Manager(LQM), which responsible for controlling the set of components [1, 3]. Also, there is a single Global Quality Manager (GQM) which carry out the study and preparation phases of the feedback loop, while the LQM is responsible for the implementation process [3].

The resources layer comprises physical (e.g., CPU or RAM) as well as virtual resources(e.g., operating system). To controlling and monitoring all resources, the Global Resource Manager(GRM) is presented. Each resource has its own Local Resource Manager (LRM), which has in-depth knowledge about its resource and the ability to steer the resource by [1, 3].

2.1.3 MQuAT combine the design time and runtime

As showed in Figure 2.2, in addition to the actual code, the developer is creating the models and quality contracts. The users interact with the system at runtime and prepare their objectives and requests. In addition to the running components, the runtime system includes a runtime model, representing the current state of the system. The objectives of the user are transformed into objective functions of an optimization program prior to optimization. Depending on the type of optimization technique used, either all users ' objectives are merged into a single objective function, or one objective function is extracted per purpose. Such objective functions and the system's runtime model are used by the runtime optimization method to produce formulations of the optimization problem for the respective technique. Finally, the system determines whether the optimal configuration differs from the actual configuration [1, 3].

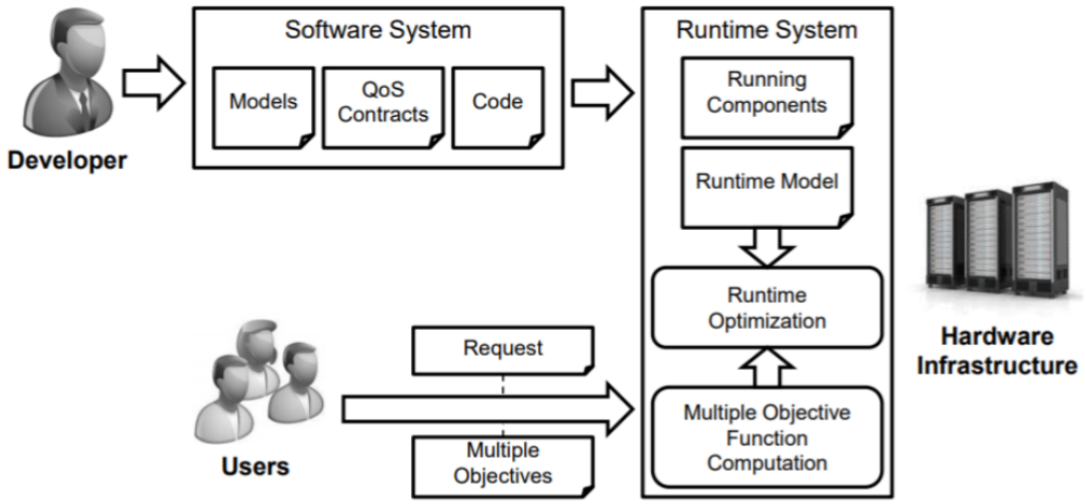

Figure 2.2 Combined structure of MQuAT

For more details about MQuAT, please read refer [3]. For this thesis, we are discussing a solver of the MQuAT problem that is based on MQuAT. Hence, detailed knowledge about MQuAT is not required.

2.2 MQuAT problem

MQuAT problem was presented in [5], and it consists of two problems:

- Resource allocation in which the mapping of software component implementation to hardware resource leads to the least cost
- Variant selection, which provides the best utility by selecting better software implementations.

To solve this problem was presented new generic metamodel. Both problems are interrelated by user requests specifying minimum requirements on the provided non-functional properties (i.e., minimum utility) while searching for a selection and mapping both to maximize utility and minimize costs. Correctness denotes that only solutions that *do not violate* the users minimum requirements are considered **valid**.

The problem to be solved is selecting variants of software components and mapping them based on user requests to suitable hardware resources.

The MQuAT problem could be described as a metamodel which consists of:

- Hardware metamodel, which consists of hierarchically structured resource types and resources as instances of these types. So the hardware model composes static awareness of resources (types) and knowledge of runtime (instances). Certain types of resources can run the software, i.e., they are valid targets for software implementation mapping. The container attribute is used to mark such types. Figure 2.3 depicted Hardware metamodel.
  In addition, a set of properties further characterize resource types. Resources specify then specific values for these properties. As an example, the resource type RAM could be defined with a property amount of memory and marked as a container

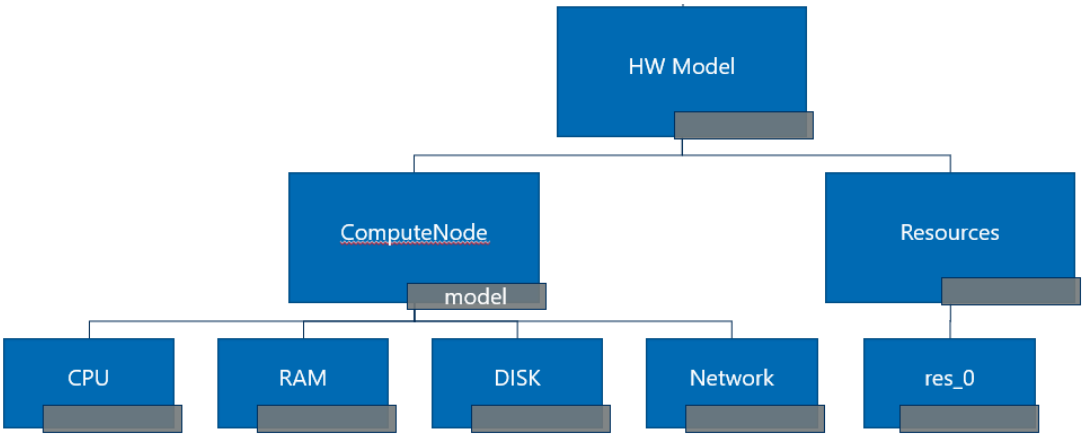

Figure 2.3 Hardware meta-model

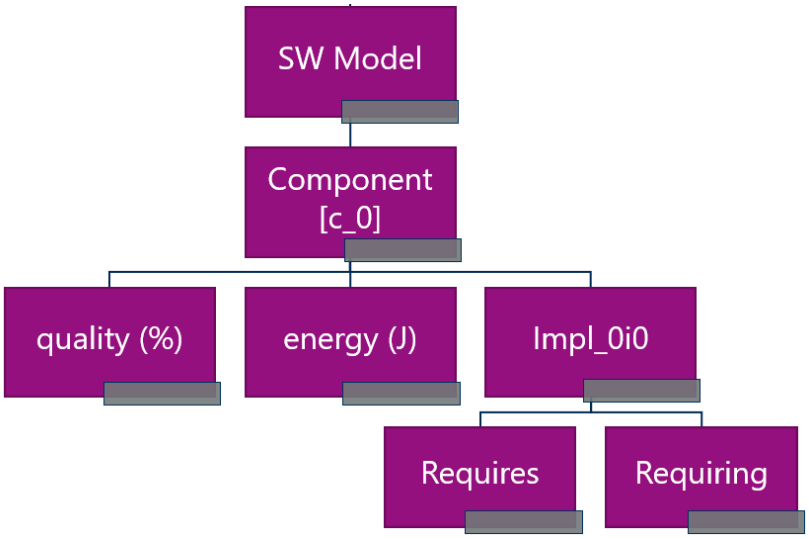

Figure 2.4 Software meta-model

- Software metamodel showed on Figure 2.4. Its main element called Component and represents some functionality. Each Component consists of implementations that provide this functionality, requiring additional components or resources to complete their work.
- The Objective specifies how to calculate a solution's objective value, i.e., for which value(s) the problem should be optimized. This selects a property to optimize for, and an objective function to define how to aggregate all values of this property.
- A Request represents a user requirement, which specifies which algorithm should be used to execute parameters and requirements. Requests contain their functional requirements by referring to a target software component, limitations on non-functional requirements (e.g., quality).

Full problem depicted on Figure 2.5

There are some constraints that are grouped in Architectural, Request, and Negotiation constraints groups.

- Architectural constraints ensure that each request is fulfilled, selecting exactly one implementation per Component and deploying no more than one implementation on one resource.

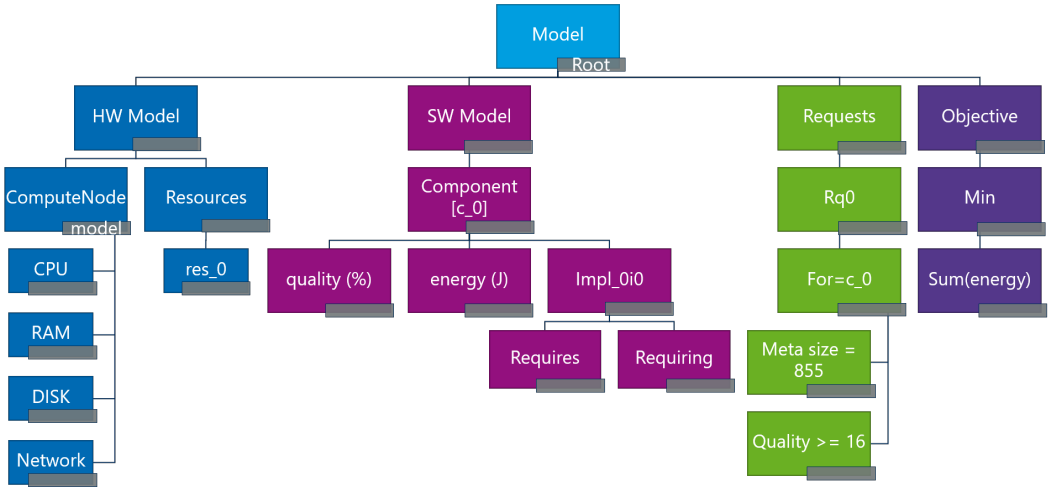

Figure 2.5 MQuAT problem model

- Request constraints ensure components are selected for each request so as to provide the requested non-functional properties.
- Negotiation constraints ensure non-functional requirements are met depending on the implementation.

There are additional constraints due to problem generation:

- structures for the software and hardware components are fixed to ensure comparability,
- computeNode that represents a regular computer hardware consist of one or more CPUs, RAM memory, disk, and a networking interface,
- software model has a simple tree structure,
- fixed branching factor of two.

Should I add parameters that describe a problem(software variants, number of requests and so on)?)

2.3 The solution of the MQuAT problem

The solution is computed by the MQuAT solver. There are many solvers:

- Simple solver, which goes step by step from one solution candidate to another.
- ILP solver, which generates integer linear programming (ILP) problem from the MQUAT problem and after that solve it.
- Random solver - tries random solution candidates.
- Simulated Annealing (SA) solver, based on the simulated annealing meta-heuristic pukhkaiev19.
- Genetic solver that uses a genetic algorithm to solve the problem.

In this thesis, we talk about Genetic solver in detail in the next sections. The solution could be represented as a tree structure. An example of the solution shown in Figure ??

It contains a list of assignments. Each assignment select one implementation of required component and map it to the resources [5]. A solution is valid if for each user request

1. an implementation is deployed for the target component,
2. for each Component required implementation is deployed,

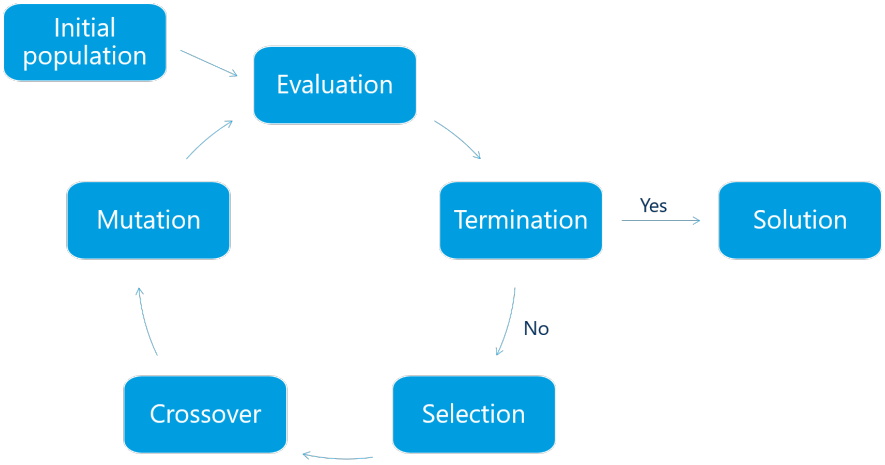

Figure 2.6 Main loop of the genetic algorithm

3. all necessary (non-functional) property clauses (including request constraints) are met,
4. at most one implementation for each resource is deployed.

If it is valid and no other solution has a better objective value, then a solution is optimal [5].

2.4 Genetic algorithm

Evolutionary algorithms are a subset of evolutionary computation and belong to set of modern heuristics-based search method. [9] Appeared as a result of the influence of the biological evolution on computer scientists. This domain contains different types. There are

- Genetic algorithm
- Genetic programming
- Evolutionary programming
- Evolution strategy

Main principle of GA looks like a loop with several steps. The first step is the creation of an initial population - a set of randomly created individuals; each of them represents one solution. The second step is Evaluation. Calculate the fitness or objective function of the current population. If the solution is founded and all requirements for the termination are fulfilled, then we get the final solution. Otherwise, select the best candidates to create a new generation. In step 4, using recombination and mutation on selected candidates, GA creates a new generation of the population. And after that, evaluate it.

GA based on different components and operators.

2.4.1 Selector

The selector is one of the most important components of GA. It selects *mu* number of individuals from the population.

There are many different selection algorithms

- NSGA2 (Non-dominated sorting based genetic algorithm)

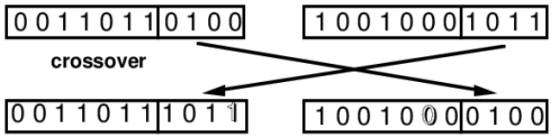

Figure 2.7 Example of the crossover between chromosome that described as a vector of bits

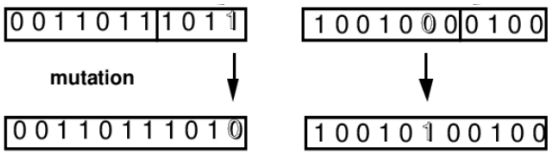

Figure 2.8 Example of the mutation between chromosome that described as a vector of bits

- SPEA2 (Strength Pareto Evolutionary Algorithm)
- NSGA3
- SPEA3
- PDE

litref
litref
litref

In this thesis, we focus on a selection algorithm only as a parameter of a genetic algorithm and use NSGA2 and SPEA2 due to software constraints of the used framework.

Will add more information about selectors

2.4.2 crossover

Crossover is an operator of a genetic algorithm that allows the recombination of two individuals by swapping some genes between them. In general, the crossover has several parameters such as

- Crossover rate - parameter that describes the probability of two chromosomes to exchange their genes.
- Crossover point - the point in which the exchange could be done.

The principle of the crossover is next. Firstly, select the crossover point. For example, a chromosome could be described as a vector of bits. Then the crossover point is the start index of bits, which were replaced by another chromosome. Secondly, swap genes between chromosomes.

2.4.3 mutation

The mutation is an operator of a genetic algorithm that changes a single gene in a chromosome. As a crossover operator, a mutation has parameters:

- Mutation rate - parameter that describes the probability of mutation.

To perform a mutation on chromosome need to do:

1. Randomly select gene which mutates
2. Change selected gene to another.

Figure 2.8 presented a simple mutation on the chromosome that described as a vector of bits.

2.5 Genetic solver

To solve the MQuAT problem using a genetic algorithm, the genetic solver was developed by Jamal Ahmad in [1]. And further improved by Johannes Mey.

This solver is based on Opt4J framework [1]. It's an open-source framework that gives the opportunity to implement a genetic algorithm for custom optimization problem by specifying several modules and classes.

To solve the custom problem using genetic algorithm, the user needs to create several things:

1. Creator is needed to create a random genotype for the initial population. In the genetic solver Creator create the genotype by creating a random solution model and transform it into a Tree Shape Genotype structure.

2. Decoder is needed to perform decoding the tree shape genotype into phenotype. The phenotype, in this case, is a Solution Model of MQuAT.

3. Evaluator calculates the objective functions of the solution. In the case of genetic solver, the Evaluator calculates two objectives:

    - Validity errors - number of violated contracts
    - Energy value - energy consumption

If your genotype can't be described as a vector, then you first need to implement:

1. Genotype

2. Crossover operator

3. Mutation operator

2.5.1 Tree Shape Genotype

Because of the problem model of MQuAT, which requires mapping of implementations to resources, in genetic solver was created Tree Shape Genotype [1]. The example of this genotype shown on Figure 2.9

The first node in this genotype contains the input request. The second node represents the mapping of the user component. In this node, Impl A-1 is selected rather than Impl A-0 of the user component. As shown in Figure 2.9, this implementation is mapped to Hardware-Resources1. Moreover, Impl A-1 also requires software components B and C that have only one Impl B-0 and Impl C-0 implementation and are mapped to Hardware-Resource5 and Hardware-Resource4, respectively.

2.5.2 Crossover operator

This operator performs a crossover between two Tree Shaped Genotypes and are performed on software implementation and hardware resource in each tree shape genotype. Figure 2.10 depicted the crossover. During this process, there is a check that ensures that the implementation and resources are not the same. If this check showed that comparing nodes are the same, then crossover goes recursively to child nodes and performs the check. If the check showed that nodes are not the same, they swap the implementation, the resource, and all lower substructure.

[1] http://opt4j.sourceforge.net/download.html

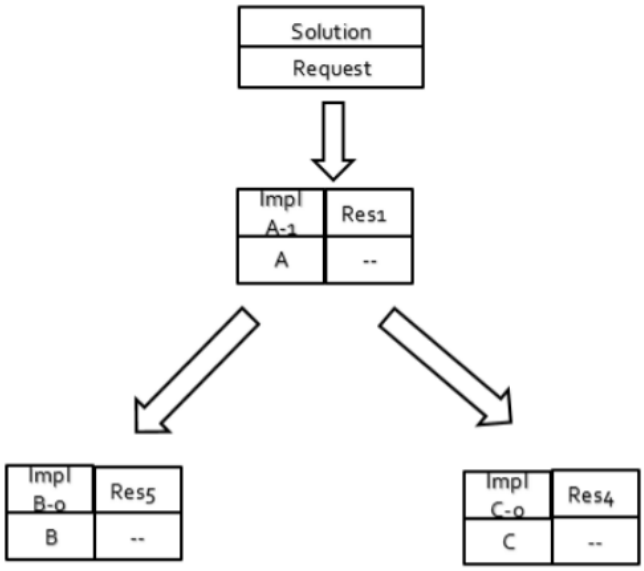

Figure 2.9 Example of the Tree Shape Genotype

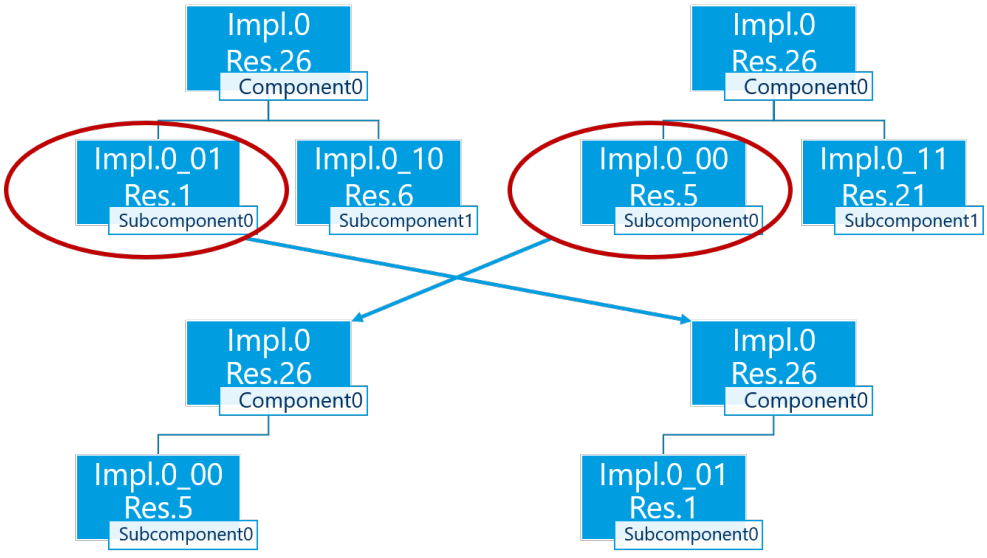

Figure 2.10 Example of crossover between two Tree Shape Genotypes

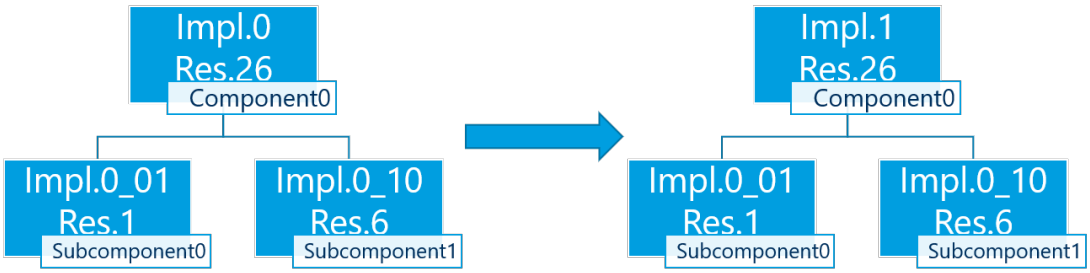

Figure 2.11 Example of mutation between two Tree Shape Genotypes

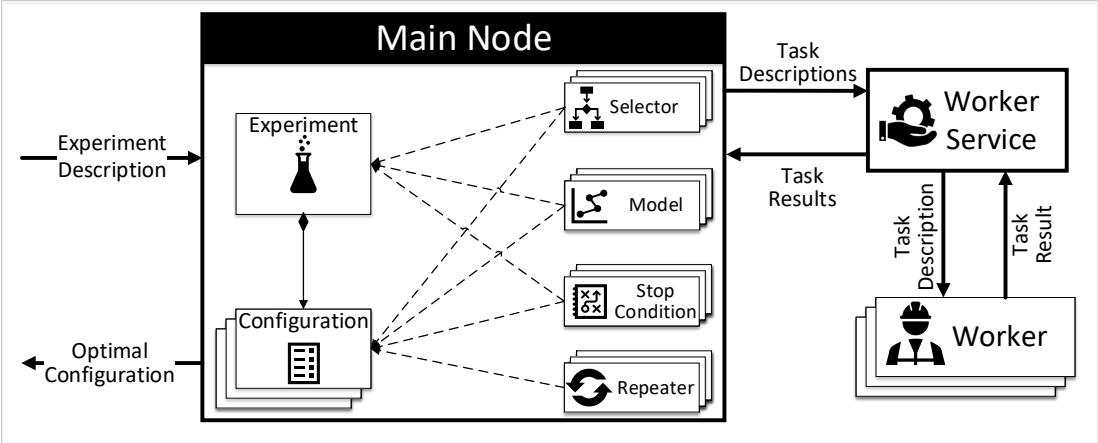

Figure 2.12 The high-level architecture of BRISE

2.5.3 Mutation operator

Mutation operation is used to randomly add new features to the tree-shaped genotype. Figure 2.10 depicts the mutation process in which with some probability to mutate current node or recursively go down to children. There are two more probabilities that represent a chance of mutation in implementation and resource accordingly.

2.6 Parameter Tuning Strategies for GA

information from literature!

will add more in next draft

2.7 BRISE

BRISE is a software product line (SPL) for parameter tuning [7]. This SPL has two conceptual parts:

- Static part describes the main flow of parameter tuning, task management, and reporting.
- Configurable part should be configured for each experiment.

Figure 2.12 shown the high-level architecture of BRISE. Configurable part consist of

- Experiment description - describes the experiment, parameters that need to tune, the objective of the experiment, and expected values to detect outliers.
- Selector - selection algorithm that is used to get the combination of parameters from the search space of all possible combinations. BRISE has two selection algorithms out of the box: Sobol sampling [8] and Fedorov's exchange algorithm [2].
- Model - predict new configuration. New Model builds after each iteration of measurement of configuration, and if Model predicts a valid result, this configuration goes to be measured in the next iteration. BRISE has several models out of the box.
- Stop condition - the Component that analyzes the results of the experiment, validates the result, and makes a decision to stop the experiment or continue. There are many different stop conditions that could work as a combination of criteria.
- Repeater - component of BRISE that decide about the number of measurement for each configuration to get need accuracy of the results.
- Worker - contains the process that BRISE needs to tune. Run this process with different configurations and returns the metric of this configuration to use it in the next model build.

All components from the configurable part could be extended by users to achieve their goals.

3 The way to get better results

In this chapter described several approaches to improve genetic solver to solve more complex problems. To do this, a benchmark measurement was made. We concentrate on solving the problem with parameters:

- variants: 10,
- depth: 2,
- requests: 15,
- resources: 5,
- timeout: 5 minutes.

. Due to the problem, the basic version of the genetic solver could not solve this problem at all. We do all this work to, firstly, achieve valid results, and then get a near-optimal solution. All measurements here and later were done at least five times. Figure ?? shown the box-plot of number of contract violation in basic version [1] of genetic solver, as shown there are no valid results cause no results with zero contract violations.

3.1 Parameter tuning as a beginning of live

new title

The first step to improve the genetic solver was a parameter optimization. To perform parameter optimization was performed a few important steps.

1. Analysis of genetic solver to find out all parameters that could be changed.
2. Adapt BRISE to work with the genetic solver.
3. Prepare Experiment description for BRISE.
4. Run BRISE to get the optimal configuration.

After deep-diving into the code of genetic solver, the list of parameters was prepared. In the basic version of the genetic solver, there are only three parameters that are changeable on call. There are:

- selectorType - type of the selector algorithm that was mentioned in section 2.4.1,
- number of generation - number of generations that performed by the genetic solver, working as a termination condition, in this thesis we use only timeout termination, so this parameter set as a huge value,

[1] As a basic version taked version from 24.09.2019, commit: 57845c126c30a1ea59cb35eb16af0bd37930dda9

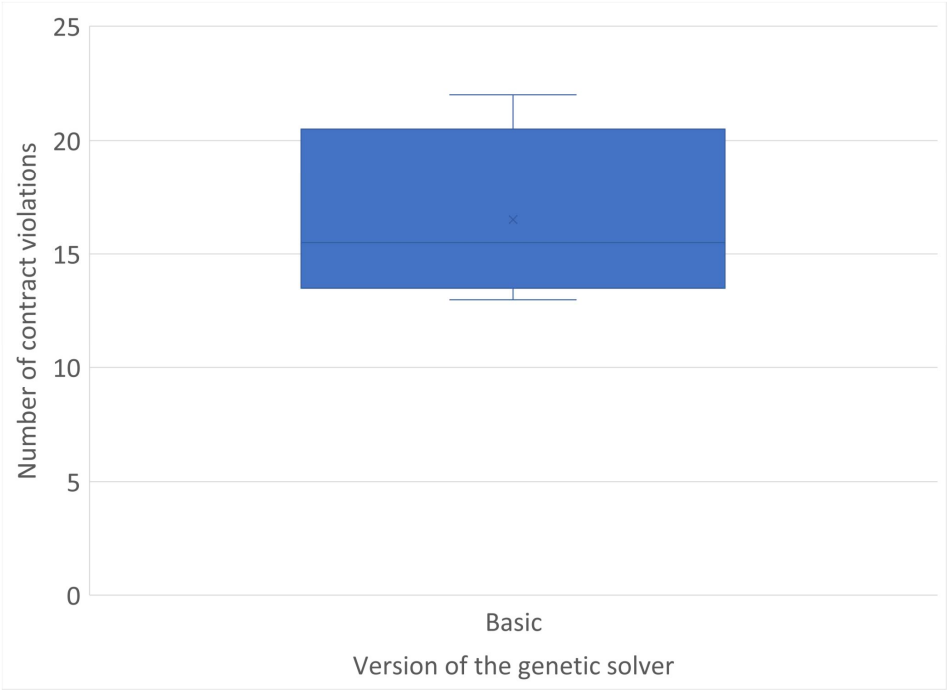

Figure 3.1

- PopulationSize - number of individuals in a population.

Adaptation of genetic solver to work into a BRISE consist of a few steps:

1. prepare executable file of genetic solver,

2. implement a method for a worker that will call the file from the previous step and returns the results with a number of contract violations and energy consumption.

Step number 3 means that the user describes parameters that need to optimize, their names, ranges if the parameter has continuous range, or all variants if categorical. The user also should set what components to use, what the minimal number of measurements of a single Configuration, etc. For parameter optimization of genetic solver, we use the experiment description shown in .

appendix or here?

When optimal configuration was founded, we use it to analyze results in a boxplot. The results showed on Figure 3.2.

Conclusions of this step:

1. All results still not valid.

2. Number of contract violations is less.

3. The distance between max and min result is smaller.

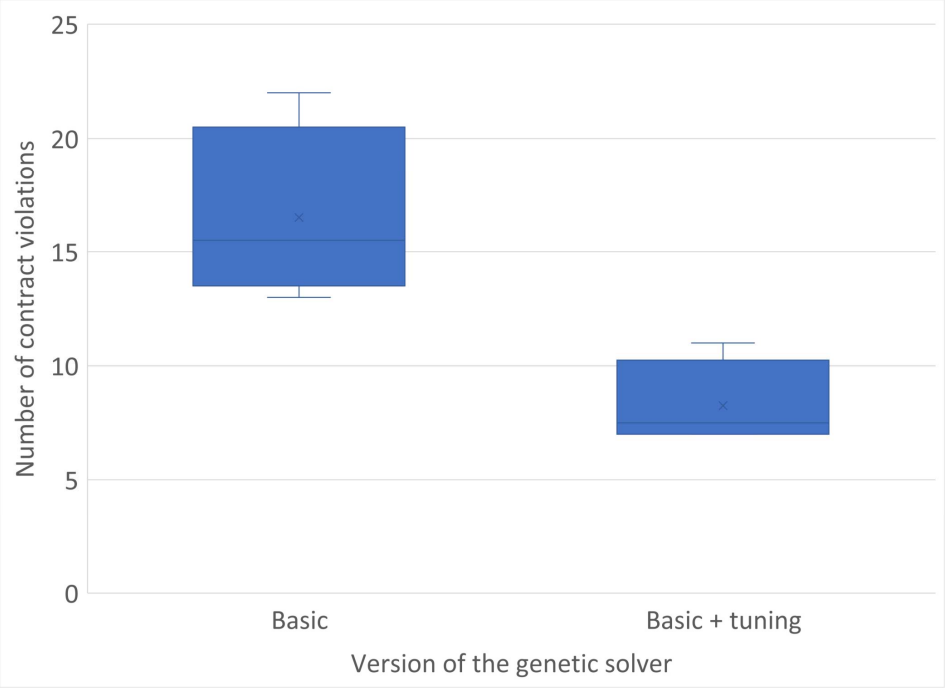

Figure 3.2

3.2 What if we could change magic numbers or Explain the code to ducks

Previous section show that even with good set of parameters genetic solver could not get valid results. After more detailed analysis of code we found out more parameters, that has static values:

- lambda - the number of new the number of new offspring individuals per generation,
- CrossoverRate - as mentioned in section 2.4.2, it describes the probability of two individuals to exchange their genes,
- mu - the number of parents, that selected by selector to create new individuals,
- MutationRate - as mentioned in section 2.4.3, it describes the probability of the individual to mutate,
- Resource Mutation Probability - the probability that during mutation mapped resource will mutate,
- CrossoverProbability - probability to do crossover inside crossover operator,
- ValidityWeight - coefficient that shows how each contract violation degrees the quality of the solution,
- SoftwareValidityWeight - coefficient that shows how each error in software tree degrees the quality of the solution,
- RandomSoftwareAssignmentAttempts - max number of attempts to set a valid software tree to individual on creation phase,
- populateSoftwareSolutionAttempts - max number of attempts to assign software component and resource to get a valid individual on creation phase.

so stupid

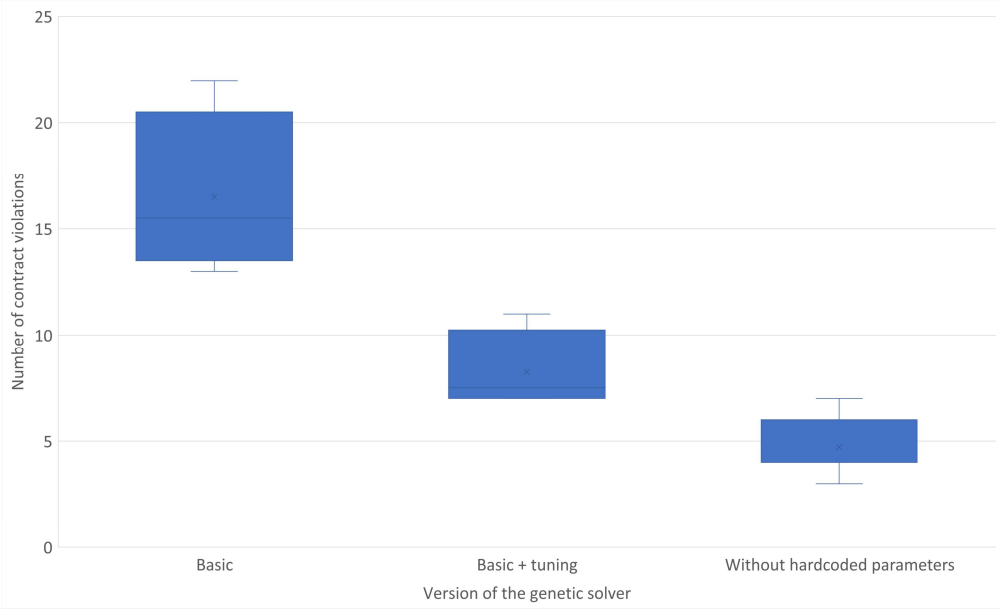

Figure 3.3 Boxplot with a number of contract violations for the genetic solver without hardcoded parameters in comparison with previous versions

To made them changeable we use *GoogleGuiceDependencyInjection* to change values of this parameters during the solver call.

Figure 3.3 shows the results of genetic solver with all parameters after optimization in BRISE. Results a bit better but still without valid results. This step confirms the fact that good values of parameters have a big influence on the results, but sometimes parameter tuning not enough to get good results.

3.3 Astrologers have announced a week of probabilities. The number of probabilities doubled.

When parameter tuning could not improve your algorithm or process, then this job must do another research field that called parameter engineering. Detailed analysis of crossover and mutation operators showed that crossover point and mutation point are pretty static.

Crossover point is defined as a node of the tree shape genotype recursively comparing the corresponding nodes of the threes by software implementation and the hardware resource for each request. If it is not located at the root of the tree, then the exchange occurs with all children.

Mutation point is defined for one random request and with probability 'MutationRate' it mutate the root of the tree shape genotype otherwise both children.

To made crossover and mutation points are more random we add more probabilities that moves them in different ways. There are:

- Crossover On Random Child Probability - chance that crossover will occur on the random child, or on both children otherwise,
- Crossover On Random Level Probability - probability that describes a chance to do crossover on random level of the tree,

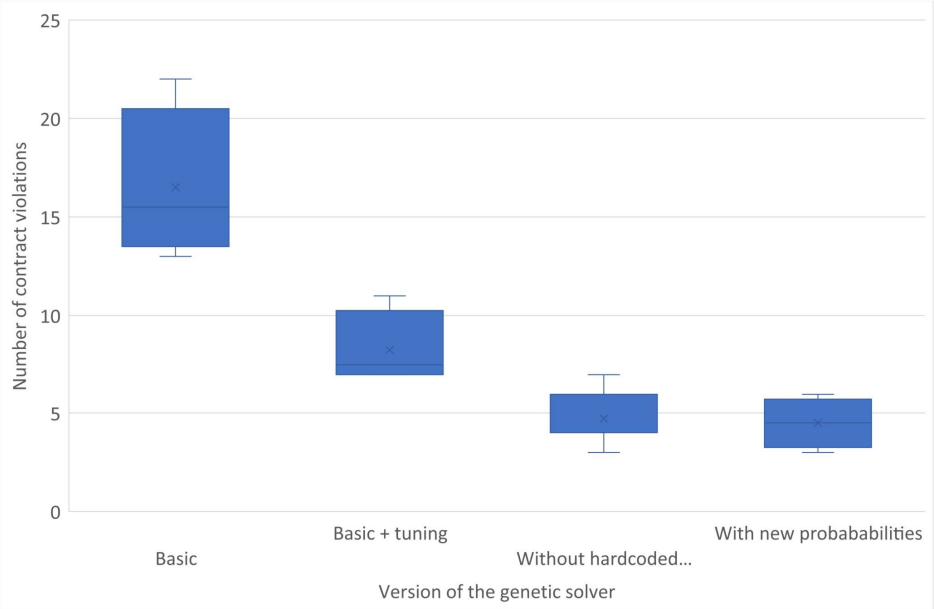

Figure 3.4 Boxplot with a number of contract violations for the genetic solver with added probabilities without tuning in comparison with previous versions

- Crossover On Random Request Probability - chance to do crossover on random request,
- Mutation On Random Child Probability - probability that describes a chance of mutation on random child,
- Mutation On Random Level Probability - chance of mutation on random level of the tree shape genotype.

After those probability were added to operators with some default values of probabilities and all previous parameters were as in basic version, the results of genetic solver was almost the same as a results from previous section (Figure 3.4).

This results show that new parameters have big influence on result as tuning. That means that good parameter engineering is important in any process. But what if we could tune new parameters together with other? Figure 3.5 shows the result that combine new probabilities and parameter tuning.

Combined approach of new parameters and parameter tuning gives the valid result. The boxplot also shows that not all results are valid. The reason for this is the local optimum and the inability to get out of it due to the fact that at a certain moment the majority of the individual in the population becomes the same.

3.4 Peppa Pig and adaptation for own life

The reason why all individuals in the population are the same is primarily due to the crossover. To create new individuals, the selector takes the best individuals. Crossover,

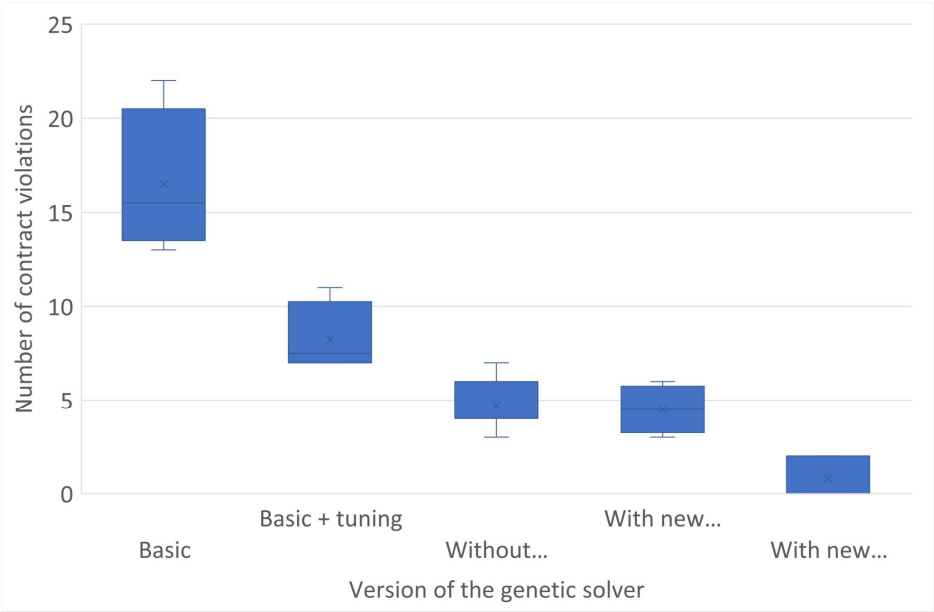

Figure 3.5 Boxplot with a number of contract violations for the genetic solver with added probabilities and tuned parameters in comparison with previous versions

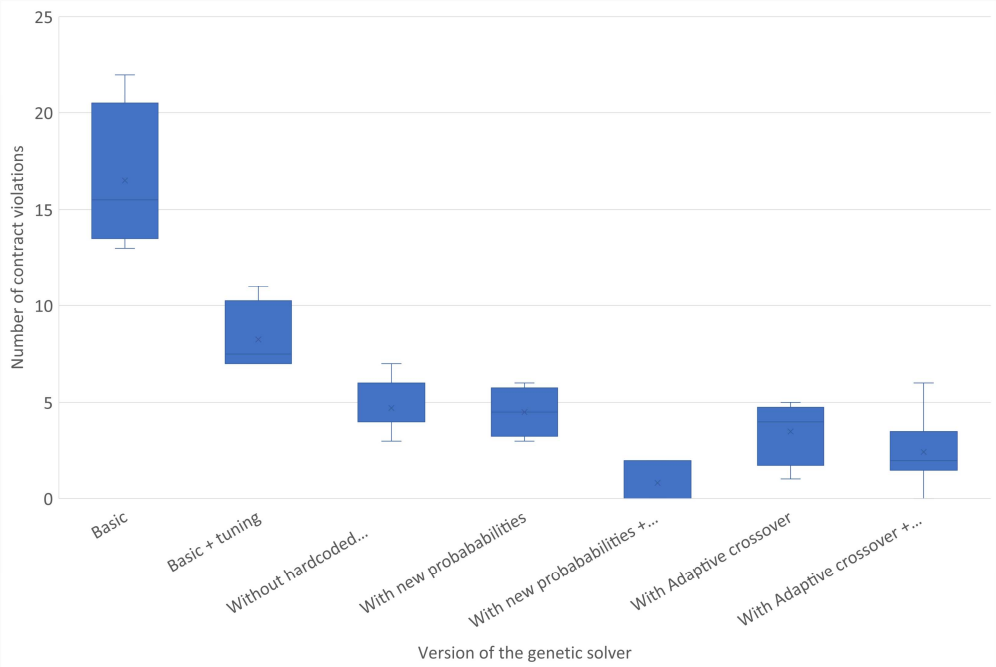

Figure 3.6 Boxplot with a number of contract violations for the genetic solver with adaptive crossover rate in comparison with previous versions

taking two decisions and performing a gene change, creates two new idividuals. If these individuals are worse, then the selector in the new iteration will drop them, and if they are close, they will take them to create new individuals. At a certain point, a situation occurs that the crossover occurs between two identical genotypes, which creates two exactly the same individuals. Which exponentially increases the number of identical elements and leads to the collapse of the genetic diversity of the population.

The idea how to solve this issue is adaptive crossover rate that change chance of the crossover depends on number of the same individuals in population. The principle of operation is as follows. After each iteration, a part of the unique genotypes in the population is calculated. If the resulting number is small, then the probability of a crossover increases, and vice versa, the probability of a crossover decreases with a small number of unique individuals.

The results of genetic solver with added adaptive crossover rate with basic set of parameters and with tuned parameters showed on Figure 3.6

As a conclusion from this section is next:

1. Adaptive crossover rate gives little improvement in comparison to previous untuned version, but results not valid.

2. Parameter tuning works good for static parameters, but for parameters that changes during the run of the genetic solver results are worse, because BRISE doesn't know that parameter could change and most important how it change.

3. Adaptive crossover rate helps to slightly delay the collapse of the diversity of individuals in the population, but this is not enough.

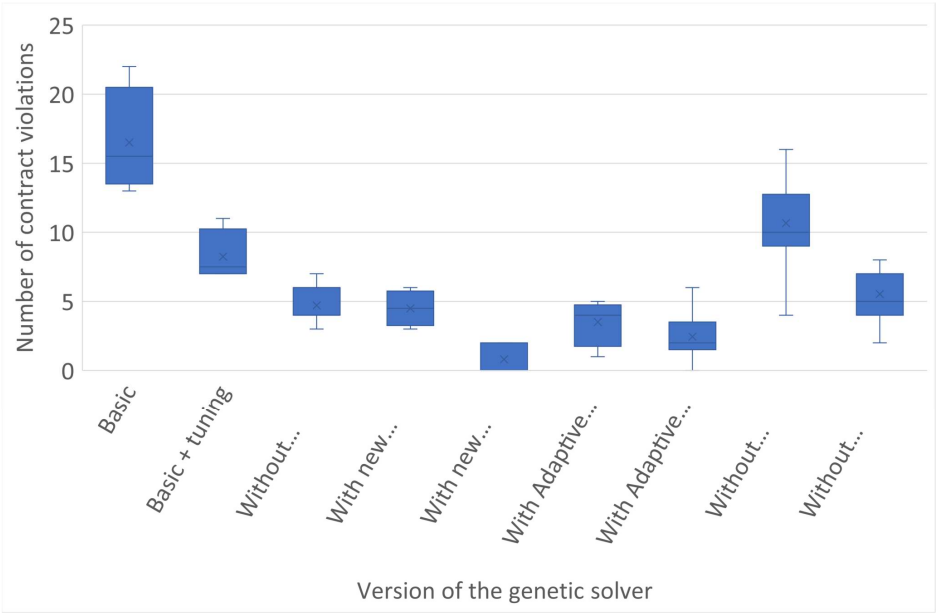

Figure 3.7 Boxplot with a number of contract violations for the genetic solver without duplicates in the population in comparison with previous versions

3.5 Unique genotypes is a God sign or placebo for genetic algorithms

Due to the fact that the adaptive crossover did not help to reduce the number of duplicates in the population, a more radical decision was made. If the same individual already has populations, then the second one cannot get there. Therefore, at any moment the population will consist of unique individuals. In this configuration of the genetic solver, the use of the adaptive crossover does not make any sense, so it was disabled and a constant value was used.

Figure 3.7 shows the results of this approach without tuning and with tuned parameters. In comparison with the basic version this approach gives better results, but in comparison with other approaches described previously the efficiency of current approach is worth.

The reasons for this are:

- on each generation best individual could give a new offspring only a few times,
- calculation speed of this approach is slower and as result less generation in specified time.

The conclusion of this idea is quite pessimistic due to fact that other approaches works better on this problem, but it may works better then others with the different problems.

4 Evaluation and Analysis

4.1 Evaluation

4.1.1 Benchmark set

Benchmark set consist of 36 problems with different sizes.
This set was tested with several versions of genetic solver

- basic
- with new parameters
- without duplicates

all parameters were unoptimized.
What I need more:

- number of runs to get valid results, or no solutions
- Using best configuration test 36 problems (for static at least)

4.1.2 Benchmark results

4.1.3 Compare with random solver

4.2 Analysis

Why it works very bad!

- Good parameter engineering
- Self-adaptive software is not the best thing to optimize!
- GA is not the best idea for this kind of problem

Plots for analysis:

- Search space view
- correlation
- distribution of two parameters
- list for combinations of 3-15
- Barplots that shows that solver could find optimal results or near optimal combined chart of validity and objective to parameter

5 Conclusion

Parameter tuning works on genetic algorithms well, but not all tasks could be solved by GA.

6 Future work

- New approaches for parameter tuning
- re-engineering of GA solver, especially genotype and operators
- other ways to avoid local optimum

Bibliography

[1] Jamal Ahmad. *A Comparative Study of Genetic Optimization Approaches in Multi-Quality Auto-Tuning*. 2018.

[2] Valerii Vadimovich Fedorov. *Theory of optimal experiments*. Elsevier, 2013.

[3] Sebastian Götz. "Multi-Quality Auto-Tuning by Contract Negotiation". PhD thesis. Saechsische Landesbibliothek-Staats-und Universitaetsbibliothek Dresden, 2013.

[4] Sebastian Götz et al. "Architecture and Mechanisms of Energy Auto-Tuning". In: *Sustainable ICTs and Management Systems for Green Computing*. IGI Global, 2012, pp. 45–73.

[5] Sebastian Götz et al. "Quality-based Software-Selection and Hardware-Mapping as Model Transformation Problem." In: *TTC@ STAF*. 2018, pp. 3–11.

[6] Sebastian Götz et al. "Towards Energy Auto Tuning". In: Oct. 2010.

[7] Dmytro Pukhkaiev and Uwe Aßmann. "Parameter Tuning for Self-optimizing Software at Scale". In: *arXiv preprint arXiv:1909.03814* (2019).

[8] IM Sobol and Yu L Levitan. "A pseudo-random number generator for personal computers". In: *Computers & Mathematics with Applications* 37.4-5 (1999), pp. 33–40.

[9] P. A. Vikhar. "Evolutionary algorithms: A critical review and its future prospects". In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*. Dec. 2016, pp. 261–265.

Statement of authorship

I hereby certify that I have authored this Master Thesis entitled *Systematic Parameter Tuning of Genetic Optimization Approaches* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 9th March 2020

Roman Kosovnenko