Swarm Intelligence Meets Rule-Based Design Space Exploration

Alexandra Anna Sólyom, András Szabolcs Nagy
Department of Measurement and Information Systems,
Budapest University of Technology and Economics
Email: solyomalexandraanna@gmail.com, nagya@mit.bme.hu

***Abstract*—In model-driven development design artefacts (e.g. source code and system configuration) are automatically generated from models. For example, a distributed computer system, which consists of multiple different hardware and software elements, can be effectively captured by an appropriate graph-like model, which can be used to generate configuration. Several modelling problems can be automatically traced back to optimization problems, such as finding the most cost effective, reliable or efficient allocation software components to hardware components. However, finding optimal solution for such systems is a major challenge, because (1) existing approaches usually operates over vectors while the problem at hand is defined by graphs; (2) besides the model, the configuration steps may also have to be optimized; and (3) for the best results, optimization techniques should be adapted to the actual domain. In this paper, we propose to integrate the bee colony optimization technique with rule-based design space exploration to solve multi-objective optimization problems in a configurable and extensible way.**

I. Introduction

Design Space Exploration (DSE) is a method for finding various system designs at design or even at runtime, which satisfy given structural and numerical constraints. Besides satisfying these constraints, DSE searches for an optimal or nearly optimal solution.

Model-Driven Rule-Based DSE operates over the model. It starts from an initial model and evolves it in each iteration with use of graph transformation rules, until it reaches one or more constraints satisfying model states. One of the advantages of this approach is that it also provides a sequence of transformations as a solution besides the design candidate itself. Furthermore, this approach is easy to integrate with model-driven development [1].

Graph transformation rules consists of two main parts [2], a graph pattern and an operation. The graph pattern defines locations of applicable transformations, through finding pattern-matching parts of the graph, while the operation determines the possible operations on these subgraphs, using previously given schemas.

Model-driven rule-based DSE can solve multi-objective optimization [3] problems. Best solution of such problems is often non-trivial. There can be more than one equally good solutions, because we have more objectives, which could be contradicting to each other. E.g. the optimization objectives of safety and cost are often conflicting, as improving safety may lead to an increased cost.

Swarm intelligence is an effective heuristic method, for finding a good solution in reasonable time. It is an adaptation of successful natural survival strategies, such as foraging of ants, bees and birds while they are looking for food sources. A common feature among swarm intelligent methods is the simplicity of participating units and the communication between them. The most used swarm-intelligent-based search algorithms are the Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO) and Bee Colony Algorithm (BCA). We found the BCA the most promising candidate for initial investigations as it is the most flexible from the three.

Section II gives insight how the multi-objective rule-based DSE works, and it also introduces the most important concepts needed to understand DSE and Multi-Objective DSE (MODSE). Section III describes swarm-intelligence-based algorithms especially the Bee Colony Algorithm. In section IV, we present our approach to solve DSE problems. Finally, section V concludes the paper.

II. Multi-Objective DSE

Models have two main types, the *metamodel* and the *instance model*. While metamodels describe the structure of models, instance models give the exact description of them. In our case, metamodels define the acceptable structures, which in most cases enable a wide variety of models. DSE-used input models belong to instance models, and they are defined in an unambiguous way [4].

As an example, consider computers and processes in a distributed system, where the metamodel defines the possible elements – computers and processes – and possible connections between them, while the instance model gives the exact number of computers and processes and how they are connected.

A graph transformation rule defines how an instance model can be modified. A transformation rule (Figure 1) consists of two sides: left hand side (LHS) is a constraint, which defines the condition and gives context to the rule while the right hand side (RHS) specifies the operation on the model. Left hand side is given by a *graph pattern*, which consists of constraints on types, connectedness and attributes. A graph pattern has a *match*, when a subgraph in the given graph has the exact structure as the pattern. A graph pattern can have multiple matches on a model [1].

As the graph pattern can have multiple matches, each rule may have multiple activations, and in most cases it is undefined, which rule should be applied. When an activation is applied, the graph structure is modified by the transformation.

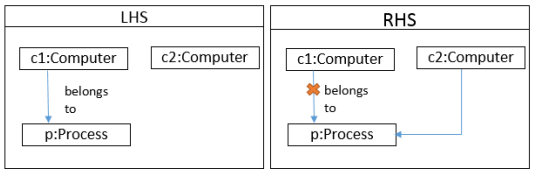

Fig. 1. Graph transformation rule for process reallocation

Well-formedness (WF) constraints (also known as design rules or consistency rules) complement metamodels with additional restrictions that have to be satisfied by a valid instance model (in our case, functional architecture model). Such constraints can also be defined by query languages such as graph patterns or OCL invariants. Ill-formedness constraints capture ill-formed model structures and are disallowed to have a match in a valid model.

For instance in Figure 2, there is a design rule, that every process has to belong to a computer. In this case, in Figure 3 there is a well-formed instance model (a) according to this rule, while in (b) there is an instance model, that is ill-formed. In the ill-formed model there is one match of the ill-formedness constraint.

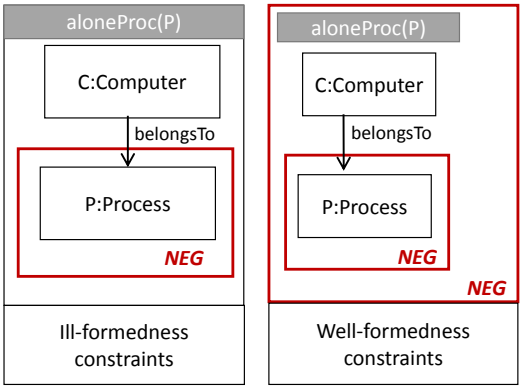

Fig. 2. Structural constraints by graph patterns

A *DSE problem* requires three input parameters: 1) an initial model, 2) a set of graph transformation rules and 3) a set of *goal constraints* captured by graph patterns. A solution of a DSE problem is a sequence of rule applications, which reaches a goal model state that satisfies all the goal constraints. These solutions are found by exploring the search space (or design space), through executing graph transformations according to an exploration strategy.

Multi-objective DSE (MODSE) incorporates objectives that express the quality of a solution. Structural (well-formedness) constraints can be also leveraged to an objective by measuring the degree of constraint violation. These objectives are either to minimize or maximize.

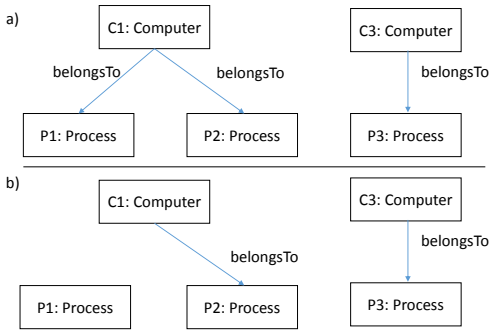

Fig. 3. Example of well-formed (a) and ill-formed instance model (b)

Objectives can be defined on both the trajectory or the model itself. While trajectory objectives measure the quality of the rule application sequence such as number or cost of operations, model specific objectives usually incorporates extra-functional objectives such as performance and reliability.

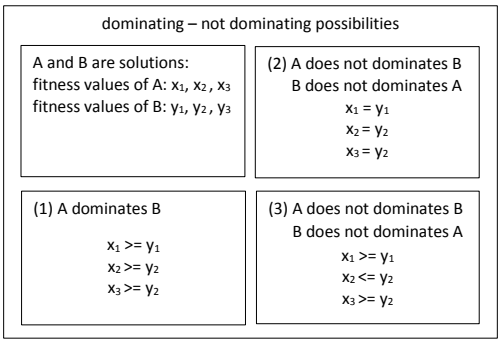

Fig. 4. Domination details x and y values are the fitness values of the solutions

While in a single-objective context solutions are easy to compare to each other, measuring and ranking (evaluating) in multi objective setting is not always obvious. For instance, if there is a computer system that has to be optimized, it is unclear whether three times faster or two times cheaper computers are the better option. It depends mostly on other aspects (size of the company, exact task, etc.), so for the same problem both can be good solutions. Therefore, the *domination function* is used in our DSE implementation to distinguish between solutions and find the best one. An example for domination can be seen in Figure 4. There are two functions, which can be the objective values of two solutions, e.g. cost, response time or safety. A solution dominates another, if at least one objective value (fitness value) is higher than the others and all other values are higher or equal. As a consequence, ordering is unambiguous, a single best solution usually cannot be determined. Instead, a Pareto front is defined, which contains all the ”best” solutions. If a solution belongs to the Pareto front, then none of it’s parameters can be increased without decreasing other parameters. In consequence, all solutions in the Pareto front dominate all other solutions, which are not

part of the Pareto front.

III. Swarm Intelligence Background

Swarm intelligence algorithms are based on modelling living groups, which successfully accomplish specific tasks, like ants, wolves or bees [5]. In these situations, a lonely animal could not survive on his own, though the whole group can. These methods are always heuristic and not aiming to find the best solution as the animals neither do it, but to find a good-enough solution in a reasonable time. Common in these packs is that an individual follows simple rules during the procedure while communicating few information to others. Such techniques can be often used for complex optimization problems, because they have good scalability and flexibility [6].

Some of the well-known swarm algorithms are the Particle Swarm Optimization (PSO), Bee Colony Algorithm (BCA) and Ant Colony Optimization (ACO). We have chosen the BCA for our initial experiments as 1) PSO is originally designed for continuous problem domains and rule-based DSE is a discrete optimization problem, 2) BCA seemed more flexible than ACO in terms of adapting guided local search exploration strategies such as hill climbing.

Bee Colony Algorithm is an often used swarm-intelligence-base algorithm, which attempts to reproduce nectar-searching methods of bee colonies. Normally, bees look for nectar in two phases. In the first phase they look for flower patches where nectar can be found. If a bee founds a patch, it goes back and performs the waggle-dance, which is a communication form between bees. Waggle-dance describes the size of the found patch and the route to it. Depending on the goodness (size, available nectar) of found patches a number of bees go out to look for the food on this patch. Then they come back and tell again how much more nectar is there.

Input parameters of the bee algorithm are: 1) the search space (problem representation, neighbourhood function), 2) the stopping criteria and 3) the size of bee population (n).

The BCA depicted in Figure 5 consists of three main phases:

- scouting phase,
- evaluation phase, and
- collection phase.

These n bees are divided into two groups. One group *(neighbourhood bees)* explores the found patches and continue to map them further, while the other group *(scout bees)* is sent out to look for new ones. Neighbourhood bees can be seen as a local search in possible optimum places while scout bees help to skip from local minimum or maximum places, and switch to a better surrounding. In the scouting phase, the first population of bees is initialized and sent out randomly to collect information. During scouting phase all n bees are scout bees which means, that they randomly explore the search space into different directions. In this phase it is important to avoid generation of similar trajectories to sample the search space in as many directions as possible. When each bee has returned then comes the evaluation phase, when patch ranking is determined, and stop condition is evaluated. Patch ranking helps to decide that which patches are worth for further exploration. If the stop criteria is fulfilled, then the algorithm can be stopped, and the best patches are selected for output. If the stop criteria is unsatisfied then we enter the loop on the right side of Figure 5. In this loop, the best collected patches are selected, and then the collection phase is started. In the collection phase, neighbourhood bees are sent out to the selected patches, and if there are more bees left (from the initial n) then these are sent out as scout bees to search for new patches.

The concrete ratio of scout and neighbourhood bees rely mostly on measuring methods [7]. In our approach, users can select the exact number of bees.

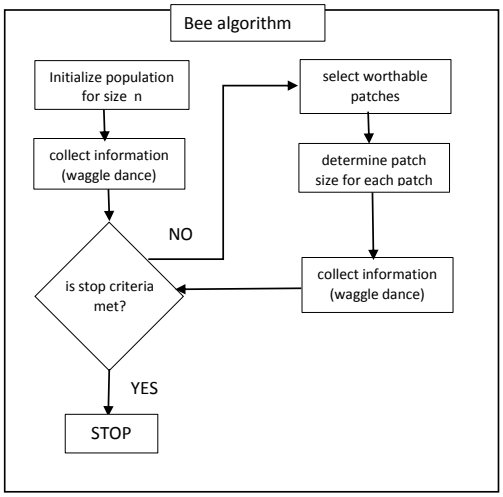

Fig. 5. Algorithm of beestrategy

IV. The Proposed Approach

The aim of our work is to use swarm-intelligence-based algorithms (namely the bee colony algorithm) for multi-objective design space exploration. While the basic challenges such as solution encoding and objective encoding were solved by Abdeen et al. in [3], adapting the bee algorithm has several other challenges, such as:

1) What is the best strategy for scout bees?
2) What is the best strategy for selecting patches for the next iteration (evaluation strategy)?
3) What is the best strategy for the neighbourhood bees?

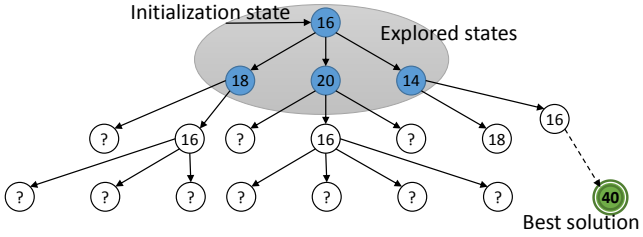

Fig. 6. A possible search space of the bee algorithm

A. Exploration Strategies for Scout Bees

The aim of the scout bees is to generate new solutions that are far enough from each other as well as previously found solutions to prevent the algorithm to stuck in local optimum. Hence, scout bees should use relatively high randomness.

While traditional approaches represent solutions as integer arrays and it is straightforward to generate such solutions, in our approach, the solutions are represented as a sequence of rule applications. We have two options for finding random nodes. The first one is to search nodes from already found patches. The advantage of this option is that it can cut down the search time by some steps, though there are situations in which scouts cannot reach every part of the graph and fail to improve the solution. For example in Figure 6, if the exploration found the three blue states with fitness values 18, 20 and 14, and the scout bees only start from the best ones, 18 and 20, then the exploration will miss the best solution depicted with green.

The other option is to search from the initial model, and go randomly to patches. Then the algorithm is able to find the green solution, though it needs more time. In this case it is hard to reach patches that are further from the initial model, and more likely to find solutions near to the initial state. The good reason behind this is that we are looking for sorted solutions. On the other hand, for scout bees it is hard to decide which direction to follow, because in this situation to reach some of the solutions they have to go through the blue states as well. As a result, we have to search more, possibly all states, which helps to find good solutions, but time-consuming. Another problem is that we have to store quite many additional data about each of the bees and their movements to avoid infinite loops. For instance, which states were good, how many times bees explored it, which other states were reached from them. If we do not store this information, then the bees can iterate through a loop, where each state is contained in the same Pareto front.

B. Evaluation Strategies

The most important decision when choosing an evaluation strategy is whether only the best or some of the worse patches should remain, and to do it in every iteration, or they should be set out only after a while. In some implementation, it is possible to sort out the wrong patches, though in our case it will not be a good idea, because the above-described hill climbing effect would come into sight. We use non-dominating sorting, which allows higher freedom for selecting the correct solutions. Non-dominating sorting means, that the algorithm separates solutions into groups according to their domination levels (number of fronts, which dominate them). In our approach, it is also modifiable how many worse solutions should be taken into consideration.

C. Exploration Strategies for Neighbourhood Bees

Neighbourhood search can be seen as a local search, which aims to discover the surrounding of a patch. It can be a hill climbing style strategy, but it gives an upper limit to the number of bees ordered to a patch. It can be a random strategy, but it is really similar to the random search in the first phase, so it has to be a combination of these two. In our approach, there are more possible strategies from which bees can choose. Some of them are similar to random search and some of them are more like hill climbing strategies with little modifications. However, each of them have some random factor to minimize the possibility of parallel-running-bees collision, which would not be a problem, but it involves unnecessary steps. Some of the usable implementations are:

- *Hill Climbing*: first, it evaluates all the neighbourhood states. Then, it finds dominating ones and randomly selects one of them, if we have enough dominated state in our list. If not, it can choose a non-dominating one that helps to avoid local minimum.
- *Simulated Annealing*: initially it steps randomly, and as time passes, the possibility of choosing a bad transformation decreases. At the end it acts like a the hill climbing algorithm.
- *Depth-First Search*: it goes through all the states till a given depth. It searches the solutions semi-randomly.

V. Conclusion and Future Work

In this paper, we proposed to integrate the bee colony optimization strategy as an exploration strategy with constrained multi-objective rule-based design space exploration. We also analysed the advantages and disadvantages of using different algorithmic configurations of the bee exploration strategy.

As for future work, we would like to measure and evaluate the effectiveness of the approach on a wide range of configurations and compare it with other exploration strategies, such as genetic algorithm and guided local search.

Acknowledgment

This work was partially supported by the MTA-BME Lendület 2015 Research Group on Cyber-Physical Systems.

References

[1] Á. Hegedüs, Á. Horváth, and D. Varró, "A model-driven framework for guided design space exploration," *Automated Software Engineering*, pp. 1–38, 2014.

[2] G. Bergmann, I. Ráth, T. Szabó, P. Torrini, and D. Varró, "Incremental pattern matching for the efficient computation of transitive closures," in *sInternational Conference on Graph Transformation*, 2012.

[3] H. Abdeen, D. Varró, H. Sahraoui, A. S. Nagy, Á. Hegedüs, Á. Horváth, and C. Debreceni, "Multi-objective optimization in rule-based design space exploration," in *International Conference on Automated Software Engineering (ASE)*, 2014.

[4] O. Semeráth, Á. Barta, Á. Horváth, Z. Szatmári, and D. Varró, "Formal validation of domain-specific languages with derived features and well-formedness constraints," *Software & Systems Modeling*, pp. 1–36, 2015.

[5] D. Karaboga, B. Gorkemli, C. Ozturk, and N. Karaboga, "A comprehensive survey: artificial bee colony (abc) algorithm and applications," *Artificial Intelligence Review*, vol. 42, no. 1, pp. 21–57, 2014.

[6] D. Karaboga and B. Akay, "A comparative study of artificial bee colony algorithm," *Applied Mathematics and Computation*, vol. 214, no. 1, pp. 108–132, 2009.

[7] D. Pham and A. Ghanbarzadeh, "Multi-objective optimisation using the bees algorithm," in *Proceedings of IPROMS*, 2007.