Research Article, 2020, Volume 7, Number 2 (pp. 141–161)

http://www.jcomsec.org

Applying Meta-Heuristics Algorithms in Model-Driven Approaches for Solving the CRA Problem

Sogol Faridmoayer [a], Samaneh HoseinDoost [a], Shekoufeh Kolahdouz-Rahimi [a,*], Bahman Zamani [a]

[a]*MDSE Research Group, Department of Software Engineering, Faculty of Computer Engineering, University of Isfahan, Iran.*

ARTICLE INFO.

Article history:
Received: 25 September 2020
Revised: 5 January 2021
Accepted: 13 January 2021
Published Online: 3 February 2021

Keywords:
Model-Driven Software Engineering, Model Transformation, Search-Based Optimization, Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO)

ABSTRACT

The Class Responsibility Assignment (CRA) problem is one of the most important problems in Object-Oriented Software Engineering. It is a Search-based optimization problem to assign attributes and methods to a set of classes such that the related class diagram has maximum cohesion and minimum coupling. Due to the large and complex search space of the problem, finding an optimal solution is a costly and challenging task. In this regard, the use of optimization approaches can be promising. In this paper, the Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO) algorithms are implemented using Model-Driven Engineering (MDE) techniques for solving the CRA problem. To evaluate the proposed approach, the effectiveness of provided algorithms is presented using models with different scales. Additionally, the results are compared with existing solutions for the CRA problem in the community. The results indicated that for large-scale models the ACO algorithm could find a much better solution in less time compared to the PSO algorithm.

© Research Article, 2020 JComSec. All rights reserved.

1 Introduction

Modeling plays an important role in all aspects of engineering and especially in software engineering. Model-Driven Software Engineering (MDSE) is a revolutionary paradigm, which uses models as a primary artifact for software development. In this way, the software engineer usually models the structure of the software system by using a class diagram and provides the actual implementation of the system by using different model-to-model and model-to-text transformations [1].

The quality of the designed diagram in the modeling phase has a direct effect on the quality of the final system. One of the main principles in object-oriented software engineering is that high cohesion and low coupling between classes make a better-designed diagram and eventually better code is generated that is easier to maintain. In this regard, the Class Responsibility Assignment (CRA) problem is a common problem in object-oriented software engineering. It is an optimization problem for assigning attributes and methods to the classes in a way that classes have the highest cohesion and the lowest coupling. In this case study, cohesion is the internal consistency of a diagram, and coupling demonstrates the independency of each class. A model consisting of

* Corresponding author.

Email addresses: `faridmoayersogol@gmail.com` (S. Faridmoayer), `s.hoseindoost@eng.ui.ac.ir` (S. HoseinDoost), `sh.rahimi@eng.ui.ac.ir` (Sh. Kolahdouz-Rahimi), `zamani@eng.ui.ac.ir` (B. Zamani)

https://dx.doi.org/10.22108/jcs.2021.125010.1057
ISSN: 2322-4460 © Research Article, 2020 JComSec. All rights reserved.

attributes, methods, and dependencies among them is the input to this case study [2].

For solving the problem, many researchers used search-based software engineering (SBSE) in past years. SBSE is one of the most important areas of software engineering, which provides (semi-)automatic optimal solutions for complex search space problems. Different algorithms such as simulated annealing, Genetic Algorithm (GA), and hill-climbing have been used to solve optimization problems [3].

In this research, the CRA problem in model-driven engineering (MDE) context is solved using the Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO) algorithms. Since the problem was defined in the MDE context, we should implement our approach in a way that is compatible with the MDE context. So we used model transformations to implement our selected search-based algorithms (PSO and ACO). The algorithms are modeled by using ATL transformation rules, and finally the effectiveness of the algorithms in solving the optimization problem is evaluated with models in different sizes.

According to MDSE principles, in MDE "everything is a model" [4]. So model transformations are also models that could be executed and apply changes in a model to transform it from one state to another. In this research, we have a source model which is a model with the structure of methods, attributes, and the relation between them, which is called Responsibilities Dependency Graph (RDG), and a target model which is an optimal class diagram. Some steps are needed to create the target model. These steps are equal to the steps of our chosen heuristic algorithms (ACO and PSO) that are implemented as some model transformation rules. In other words, the only way for transforming a source model to the target model in our case is using model transformation rules. Briefly, the main contributions of this paper are:

- Modeling and implementing ACO and PSO algorithms with the ATL model transformation language
- Presenting a general reusable framework to find an optimal output for CRA problem using the ACO algorithm
- Presenting a general reusable framework to find an optimal output for CRA problem using the PSO algorithm

The remainder of this paper is organized as follows. Section 2 presents an overview of the background concepts. Section 3 introduces the CRA problem and details of the proposed processes in this research for solving the problem. In Sections 4 and 5, the related work and evaluation of the research are explained. Finally, Section 6 concludes the paper and presents future works.

2 Background

This section introduces the background concepts of this research. Firstly, Section 2.1 introduces SBSE in MDE. Section 2.2 explains the meta-heuristic algorithms used in this study, *i.e.* PSO and ACO; and in Section 2.3 the structure of ATL transformation language will be presented.

2.1 Search-Based Software Engineering in MDE

SBSE enables (semi-)automatic optimal solution to the complex search space problems, by applying different types of algorithms, such as meta-heuristic algorithms. The problem can occur in several parts of software development, such as transformation, evaluation, analysis, and model testing. Although there are not any certain algorithms producing optimal solutions, there are some practical techniques for increasing the quality of such problems [2]; Particularly, SBSE is applied in the context of MDE for model transformation. The model transformation aims to create the target model from the input model using different types of rules; however, there is no universal solution for model transformation problems since most of the approaches for solving these kinds of problems are related to the source and target meta-models [5].

Optimization methods can be used to solve complex problems with infinite search space. The main goal of using SBSE in MDE is consolidating search methods with model-driven techniques [6]. In the following, two meta-heuristic algorithms, including PSO and ACO, are introduced to solve the CRA problem by using MDE techniques.

2.2 Population-Based Meta-Heuristic Algorithms

Meta-heuristic algorithms are an ingenious way that is capable of searching for the solution space and finding high-quality solutions. Population-based meta-heuristic algorithms contain many mutual concepts. These algorithms can be defined as a repetitive process in a population (group) of answers. In such algorithms, the initial population is created first. Afterward, the new group of answers is selected through various methods and will be merged with the current population. The periodical search is stopped when the termination condition is fulfilled [7].

Swarm Intelligence algorithms are a type of

population-based algorithms that have taken inspiration from natural behaviors of species such as ants, bees, and birds. One of the most important characteristics of these algorithms is indirect coordination among different parts of the algorithm. Ant colony, bee colony, and particle swarm optimization meta-heuristic algorithms are the most successful swarm intelligence algorithms [8]. In this research, PSO and ACO algorithms have been applied to tackle the challenges of the CRA problem.

2.2.1 Particle Swarm Optimization (PSO) Algorithm

PSO is one of the population-based meta-heuristic algorithms, which is inspired by swarm intelligence and is based on the social behavior of birds. This algorithm was introduced by Kennedy and Eberhart in 1995, based on the behavior of crowd particles like groups of crows. In a group of crows, one of the crows is the group leader and has the best position, while the rest of the crows try to stay as close as possible to the leader with respect to their neighboring crow's positions. If during this, a crow can get close to the leader's position relative to other crows, it is appointed as the leader of the group [9]. A swarm of particles is distributed in the search environment. Some particles will obtain a better position than the others. Thus, based on crowd particle behaviors, other particles try to get closer to the better-positioned particles. At the same time, the positions of the better particles are also constantly changing [10]. In this method, changing the positions of the particles are based on the particles' experience in earlier movements and its neighboring particles. Each particle is aware of its priority compared to other particles and the group [9].

To explain PSO, we consider a union of N particles flying in D-dimensional space. Each particle i is a candidate solution for the problem and is represented by the vector x_i in the search space. Each particle has a specific position and velocity that represents its direction and velocity of flight in a D-dimensional search space represented by d index in the following equations. Particles' success during the flight affects the behavior of other particles. Each particle changes its position x_i constantly to move toward optimized solutions using the following factors [7].

- The best position observed by the particle itself ($pbest_i$) which is represented as $p_i = (p_{i1}, p_{i2}, ..., p_{id})$
- The best position observed by the whole population ($gbest_i$) which is represented by $p_g = (p_{g1}, p_{g2}, ..., p_{gd})$

The vector $(p_g - x_i)$ gives the difference between the current position of the particle i and the best position in the neighborhood. A neighborhood must be defined for every single particle. This neighborhood represents the collective effect of the particles. A method for defining the neighborhood is called *gbest*.

A leader is selected to guide the particles' search toward better positions in the search space. In each iteration, each particle will perform the following operations [7]:

- Updating the velocity: the velocity for each particle at time t is defined as follows:

$$v_i(t) = v_i(t-1) + \rho_1 c_1 (p_i - x_i(t-1)) + \rho_2 c_2 (p_g - x_i(t-1)) \quad (1)$$

  Where ρ_1 and ρ_2 are two random variables on the interval $[0, 1]$ and constants c_1 and c_2 represent learning factors. These constants show the attraction of each particle towards itself or its neighbor's positions. c_1 is the cognitive learning factor that represents each particle's attraction towards its position. c_2 is the social learning factor and shows the level of attraction of each particle towards its neighbor's positions.
- Updating the position: Each particle updates its position along with itselfs direction in the space:

$$x_i(t) = x_i(t-1) + v_i(t) \quad (2)$$

- Updating the particle's best-known position: Each particle updates its best-known position if the current position x_i was better than the $pbest_i$ then $p_i = x_i$.
- Updating the swarm's best-known position: The best position found in the population is updated if at least the current position of one particle was better than the *gbest* then $g_i = x_i$.

Therefore, in each iteration, each particle changes its position concerning the experience of neighboring particles and itself.

2.2.2 Ant Colony Optimization (ACO) Algorithm

ACO is one of the optimization algorithms and has taken inspiration from the way ants work to obtain food. This algorithm was presented for the first time by Dorigo et al. [12] as a solution for hard optimization problems.

The behavior of ants for finding food is shown in Figure 1. They follow the shortest path from their nest to the food as groups. The experiments on ants show that despite existing multiple paths with different lengths, ants follow the shortest one after a little inconsistency (a few minutes), and this ratio

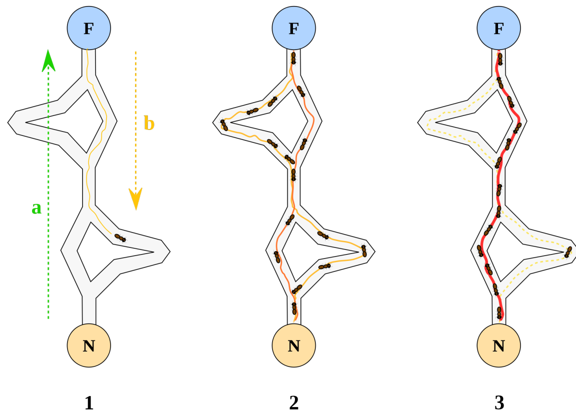


Figure 1. Ant Behavior in Goss Experiment [11].

increases with the increase of length differences [13]. Ants release a chemical substance called pheromone in the followed paths, and when they reach a crossroad, they choose a path based on the pheromone rate. Therefore, the probability of rechoosing the selected path is high. The repetition of this action will result in the accumulation of higher rates of this chemical on the shortest path. As a result, the mentioned path will be selected most often. This simple idea is used to finding an appropriate solution for hard optimization problems.

It is possible that ants do not choose the shortest path in the first step; Therefore, the concept of evaporation is needed. As a result, the pheromone gets gradually evaporated, and the ants are prevented from choosing the wrong path.

Listing 1 shows the general ant colony algorithm. In the initialization step, parameters including initial pheromone quantity are regulated.

Listing 1: Generic Algorithm of ACO.

```
Initialize including the pheromone trails and
    evaporation rate
Repeat
  FOR each ant Do
    Solution construction using pheromone trails
    Evaporation
    Reinforcement
  Until stopping criteria
OUTPUT: Best solution found or a set of solutions
```

In each step, the ant chooses a path based on the pheromone level and heuristic information. For example, in the traveling salesman problem, the distance between cities can be considered to be heuristic information. The following formula shows the probability of moving from state i to j. S is the symbol of untraversed states in the current iteration. Furthermore, α and β are positive real parameters whose value determine the relative importance of pheromone versus heuristic information.

$$P_{ij} = \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}} \quad i \in [1, N], j \in S \tag{3}$$

After the ants travel a path and find a solution, the level of pheromone in that path is updated in two steps. First, the level of pheromone in the path is updated based on the quality of the traveled path, and then the level of the pheromone will be reduced by a constant called evaporation rate. The following equations represent the functions of pheromone's updating. Δ is the amount of released pheromone by an ant in each iteration, and τ_{ij} represents the current amount of pheromone. The evaporation step is expressed in Equation (5) where ρ shows the evaporation rate, which is regulated in the initialization step.

$$\tau_{ij} = \tau_{ij} + \Delta \tag{4}$$

$$\tau_{ij} = (1 - \rho)\tau_{ij} \tag{5}$$

2.3 The ATL Transformation Language

MDSE is an approach to cope with the complexity of software development by increasing the abstraction level of the studied system. Model Transformation, the heart and soul of MDSE, being used to transform one or more input model(s) to one or more output model(s) in-order to generates the final system automatically [1].

A transformation definition is a set of transformation rules that describe how a source model can be transformed into the target model. The source and target models conform to the same or different meta-models (AKA modeling languages in MDSE). A transformation rule is a description of how one or more constructs in the source meta-model can be transformed into one or more constructs in the target meta-model [14].

ATL is one of the commonly used languages for model transformation, which follows the declarative style for performing the transformations. In the descriptive condition, model transformation is based on certain rules, each of which identifies a set of elements of the model. The imperative structure in ATL is defined in different parts of the transformation specification. An instruction block can be added to describe rules for assigning values to the target model. ATL is compatible with every device that has

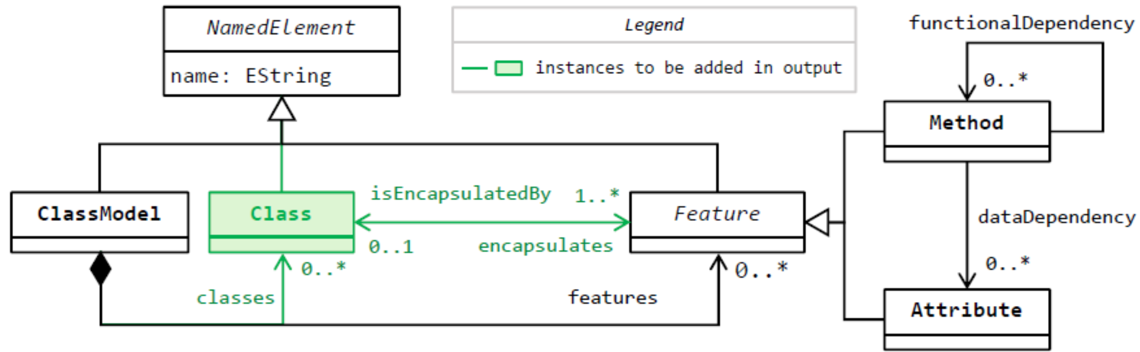


Figure 2. Class Diagram Meta-Model [5].

an ATL transformation engine. Eclipse [1] is an integrated development environment with ATL plug-in availability option. The Source and Target models in ATL can be expressed in XMI [2] format, and source and target meta-models can be expressed in XMI or KM3 [3] formats [15].

Listing 2 is a simple example of ATL model transformation. In this transformation, the input model IN which conforms to the meta-model MM-IN transforms to the output model OUT which conforms to the meta-model MM-OUT. Rule1 transforms the input model to the output model. In this Rule, each element of Type1 in the input model that has the cond1, transforms to a similar element in the output model. The Prop's property of the input's element copies to the corresponding Prop's element in the output model.

Listing 2: A Simple Example of Transformation in ATL.

```
module moduleName;
create OUT: MM-OUT from IN: MM-IN;
rule Rule1{
  from
    V1: MM-IN!Type1(cond1)
  to
    V2: MM-OUT!Type1(
      Prop <- V1.prop
    )
}
```

3 The CRA Problem in Model-Driven Engineering and Its Presented Solutions

The CRA case study was introduced in the 9th Transformation Tool Contest (TTC 2016). In this case study, the structure of methods, attributes, and the relation between them, which is called Responsibilities Dependency Graph (RDG) are provided. The main goal of the presented case study is to transform the Responsibilities Dependency Graph to a class diagram with the highest cohesion rate and the lowest coupling rate. This transformation is considered to be an endogenous transformation as both the source and the target models are instances of the same meta-model [5].

Figure 2 presents the class diagram meta-model. The Responsibilities Dependency Graph is a subset of this meta-model, which contains several attributes, methods, and relationships among them. The Responsibilities Dependency Graph will be transformed into a class diagram by placing these attributes and methods inside an entity. The entity and its relations have been represented with green color in Figure 2 [5].

The main challenge, in this case, is to study the proper implementation of the transformation rules and create a qualitative class diagram. A class diagram with the correct number of classes is selected, and the appropriate attributes are assigned to the appropriate classes. Creating proper coordinations among transformation rules is also needed too. The presented issue is a very complex task and creates a very large search space. Output class diagram must have some properties [5]:

- All features of the input model must be encapsulated in a class.
- The output diagram should have no empty classes.

As mentioned earlier, the CRA case study is an optimization problem with two main objective functions: minimizing the coupling rate and maximizing the cohesion rate. This means that CRA is a multi-objective optimization problem. The quality of the produced class diagram is based on CRA-Index. This CRA-Index formulates cohesion and coupling rate

[1] https://eclipse.org/
[2] XML Metadata Interchange
[3] Kernel Meta Meta-Model

in addition to the relationship between them. By maximizing these criteria, the quality of the class diagram will reach its highest level. Equation (6) shows the CRA-Index formula [5].

$$CRAIndex = CohesionRate - CouplingRate \quad (6)$$

$$\text{CohesionRatio} = \sum_{c_i \in \text{Classes}} \frac{MAI(c_i, c_i)}{|M(c_i)| \times |A(c_i)|} + \frac{MMI(c_i, c_i)}{|M(c_i)| \times |M(c_i) - 1|} \quad (7)$$

$$\text{CouplingRatio} = \sum_{c_i, c_j \in \text{Classes}} \frac{MAI(c_i, c_j)}{|M(c_i)| \times |A(c_j)|} + \frac{\text{MMI}(c_i, c_j)}{|M(c_i)| \times |M(c_j) - 1|} \quad (8)$$

$$MMI(c_i, c_j) = \sum_{\substack{m_i \in M(c_i) \\ m_j \in M(c_j)}} DMM(m_i, m_j) \quad (9)$$

$$MAI(c_i, c_j) = \sum_{\substack{m_i \in M(c_i) \\ a_j \in A(c_j)}} DMA(m_i, a_j) \quad (10)$$

$$DMA(m_i, a_j) = \begin{cases} 1 & \text{if there is a dependency between method } m_i \text{ and attribute } a_j \\ 0 & \text{otherwise} \end{cases} \quad (11)$$

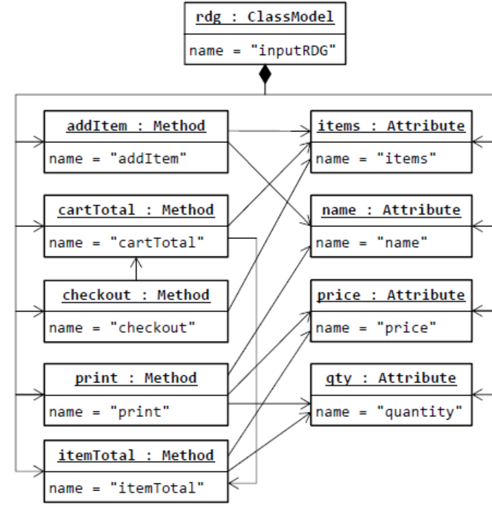
$$DMM(m_i, m_j) = \begin{cases} 1 & \text{if there is a dependency between method } m_i \text{and } m_j \\ 0 & \text{otherwise} \end{cases} \quad (12)$$

3.1 A Running Example for the CRA Case Study

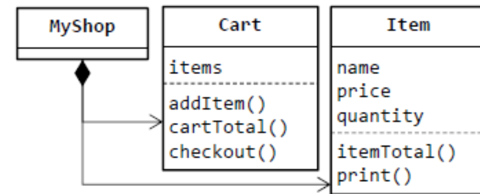
For a better understanding of the problem, a running example is presented in this section. As mentioned, the RDG model is the input model in this case study. The main goal is finding an optimal output model in a class diagram form. Minimizing the coupling and maximizing the cohesion are the objectives. Figure 3 represents an example of the CRA case study.

Figure 3a is the RDG input model. The input model shows attributes, methods, and the relations among them. None of these attributes and methods belong to any classes. As a result, the purpose of this case study is to allocate all of these attributes and methods to the classes of a class diagram with the highest Ratio of the CRA-Index objective.

Figure 3b is the output class diagram. All of the attributes and methods are encapsulated in the classes of a class diagram in a way that the CRA-Index maximizes. Figure 3c shows the CRA-Index calculations based on Equation (6).



(a) RDG Input Model.



(b) Class Diagram Output Model.

	Cart	Item
MAI(c_i, c_i)	3	5
MMI(c_i, c_i)	1	0
CohesionRatio	1.1667	0.8333
MAI(c_i, c_j)	1	0
MMI(c_i, c_j)	1	0
CouplingRatio	0.4444	0
∑ CohesionRatio	2	
∑ CouplingRatio	0.4444	
CRA-Index	1.5556	

(c) Calculating the CRA-Index for the Output Model B.

Figure 3. Example of the CRA Case Study [5].

Briefly, the example shows how the ordering of the attributes and the methods can optimize the CRA-Index Rate. In this paper, we solve the CRA problem by using PSO and ACO algorithms. To do this in the MDE context, we implement those algorithms by model transformations. Sections 3.2 and 3.3 show how the CRA problem solved using PSO and ACO algorithms implemented via MDE approaches. Additionally, the source code of the presented solutions are located in GitHub [4].

[4] https://github.com/Ariyanic/CRA.git

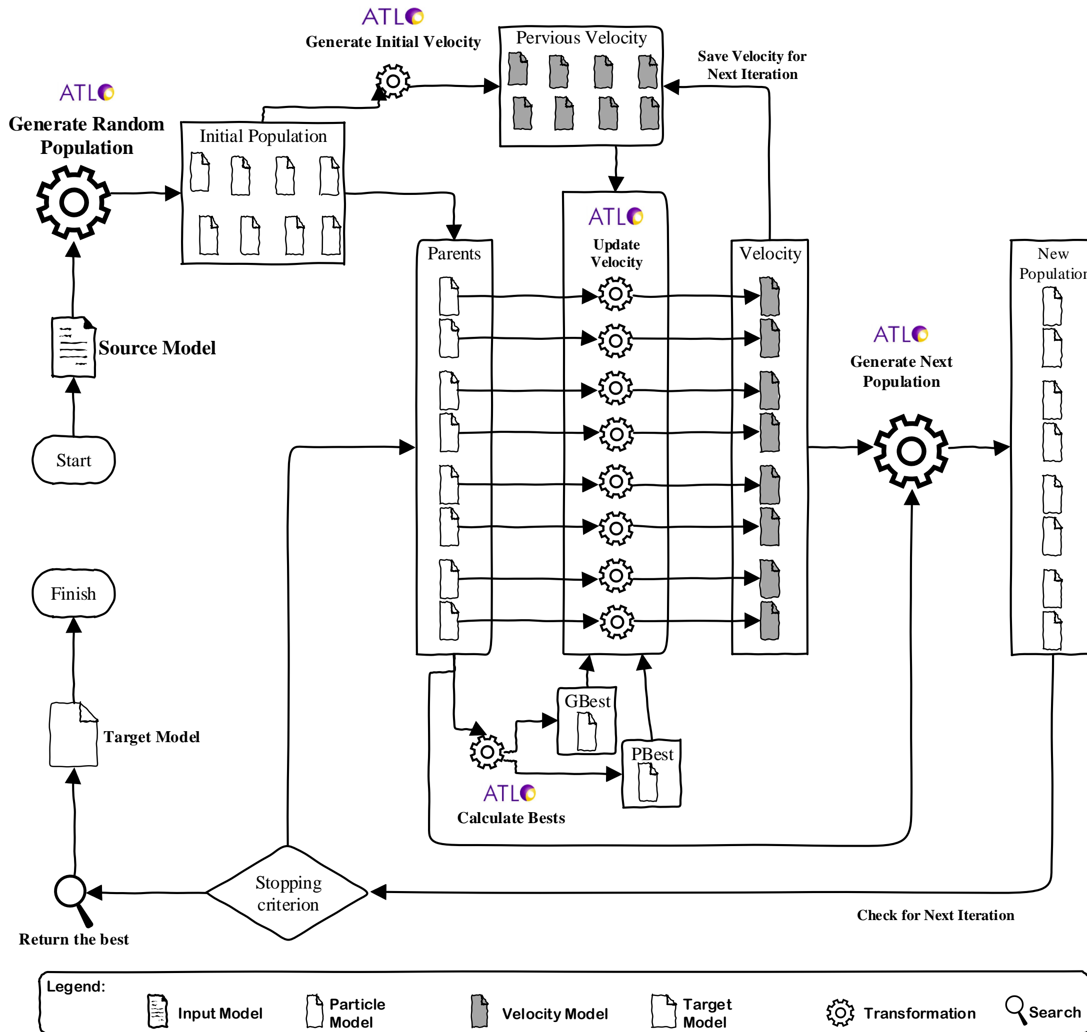


Figure 4. The Model-Driven Approach for Solving the CRA Problem Using the PSO Algorithm.

3.2 Solving CRA Problem Using PSO Algorithm

In the following section, a model-driven approach for solving the CRA problem using the PSO algorithm is presented.

As mentioned in the previous section, in the PSO algorithm, an initial population will be created in the first step. Each particle in this population is searching for the optimal point in the search space; thus, the particles are moving with an initial velocity [16]. The velocity of each particle i at each moment t is determined by Equation (1). For choosing proper values for the parameters that are used in Equation (1), the following concepts are considered:

- Exploration: The ability for finding the optimized global solution
- Exploitation: The ability for finding the optimized local solution

Constants c_1 and c_2 can be initialized in an interval of [0-2]. If these constants increase, exploration will increase, but whatever the constants get lower, exploitation will increase. To find an optimized solution, both of the following concepts are needed. As a result, the following initializations are chosen to reach the optimized solution:

$$\rho_1 = 1 \qquad c_1 = 1 \qquad \rho_2 = 1 \qquad c_2 = 1$$

Following that, the new position of each particle is obtained by Equation (2).

Figure 4 shows the PSO algorithm process for solving the CRA problem. As shown in this figure, first of all, an initial population is generated from the source model. The source model contains several attributes and methods with dependencies between them. These features do not belong to any classes. Using *randomPop* transformation, each feature is en-

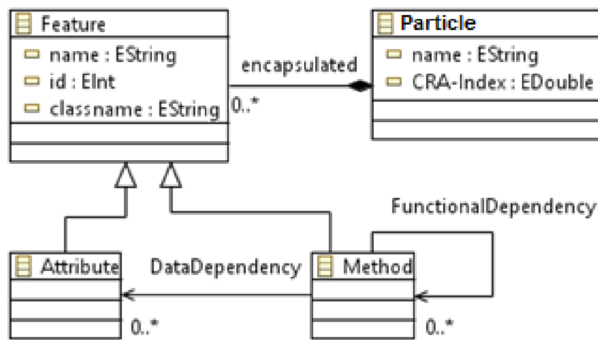


Figure 5. The Particle Meta-Model.

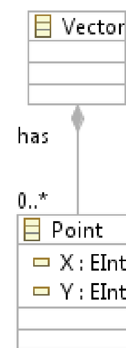


Figure 6. The Velocity Vector Meta-Model.

capsulated by a class randomly, and in this way, the initial population is generated. In other words, each particle in the population is a random assignment of the features into the classes (as the same Chromosome in Genetic algorithm [17]). Figure 5 shows the structure of a particle in our proposed approach and listing 5 (in appx. A.1) shows *randomPop* transformation rules.

After generating the initial population, the CRA-Index criteria [5] of each particle is measured, and the particle with the greatest CRA-Index is marked as the *gBest* of the swarm in that iteration. Additionally, as mentioned earlier, $pBest_i(t)$ will be the best position (the greatest CRA-Index) of particle(i) until iteration(t). For the first iteration, *pBest* of each particle will be itself, and for the other iterations, it will be the position of the particle that has the greatest CRA-Index until that iteration.

In the PSO algorithm, a velocity vector is used for updating the position of each particle in each iteration. The vector contains several points that make the new position of each particle. In other words, a vector model should conform to a meta-model, as shown in Figure 6.

According to the meta-model, each vector contains 1 to n points, in which n is the number of features in the source model. Each point has two Integer numbers, X, and Y. X is equal to the *id* of features in a particle, and Y is equal to the *id* of classes that corresponding feature should be encapsulated by it.

In each iteration of the PSO algorithm, a velocity vector is generated for each particle (according to Equation (1)). This is done by the *randomVector* transformation rule (listing 3).

The *testRand()* method generates an Integer number between 1 to n, which n is the number of features in a particle. According to Equation (1), for calculating the difference between two particles, *i.e.* $p_i - x_i(t-1)$ or $p_g - x_i(t-1)$ a transformation rule is written as shown in listing 4.

Listing 3: *randomVector* Transformation Rules.

```
-- @atlcompiler emftvm
module randomvector;
create OUT1 : velocityVector from IN: particleCRA ;

helper def : fSize : Integer = particleCRA!Feature.
     allInstances().size() ;

helper def : testRand(i : Integer) : Integer =
let rnd : "#native"!"java::util::Random" = "#native"!"
     java::util::Random".newInstance() in rnd.nextInt(i
     ) + 1;

rule randomVector{
  from
    ft : particleCRA!Feature
  to
    p: velocityVector!Point(
      X <- ft.id ,
      Y <- thisModule.testRand(thisModule.fSize)
    )
}
```

Listing 4: Calculating Differences Between Two Particles.

```
-- @atlcompiler emftvm
module computeVelocity;
create OUT: velocityVector from IN1: particleCRA, IN2:
     particleCRA ;

rule particle2Velocity{
  from
    ft1 : particleCRA!Feature in IN1, ft2: particleCRA!
         Feature in IN2
    (ft1.id=ft2.id and ft1.classname<>ft2.classname)
  to
    p: velocityVector!Point(
      X <- ft2.id ,
      Y <- ft2.classname.substring(6, ft2.classname.
           length()).toInteger()
    )
}
```

The transformation rule compares features of two particles peer to peer and saves the features of the

Figure 7. The Model-Driven Approach for Solving the CRA Problem Using the ACO Algorithm.

better particle that are not the same as the other one into a Vector model. In this case, the result of $p_i - x_i(t-1)$ will be saved into a vector named *vpbest* and the result of $p_g - x_i(t-1)$ will be saved into another vector named *vgbest*. The process is done for all particles of the current population except *gBest*, because *gBest* is the swarm's best-known position that all other particles should move towards it. In other words, a particle should move towards another only if the target particle was better particle. In the case of *gBest*, there was no particle better than it, in the current population. Therefore, *gBest* should be motionless, and other particles move towards it.

The next step is that $v_i(t)$, *vpbest*, and *vgbest* concatenate to each other to make $v_i(t+1)$ (*i.e.* velocity vector of particle *i* at time *t*). This is done by a transformation module (listing 6 in appx. A.1) that gets two vector model and concatenates them into a new vector. The transformation module calls two times in a Java file: first, *vpbest* and *vgbest* concatenate to each other and then the result concatenate with $v_i(t)$ (the vector obtained from transformation rule in listing 3).

After generating $v_i(t+1)$, the velocity vector should be applied to the current population to move them to the new positions (Equation (2)). This is done by updateParticle transformation module (listing 7 in appx. A.1).

As shown in listing 7, in *newClassID* helper, if there was at least one point in input velocity vector ($v_i(t+1)$ in Equation (2)) that its *X* was equal to one of the feature's id in input particle ($x_i(t)$ in Equation (2)), then the point that was belonged to *vgbest* (the last point with the related *X*) is selected for changing the position of the input particle and generating new particles with new positions.

After generating new particles, the *pbest* is measured for each member of the population, and the process is repeated until satisfying the termination condition (the specified number of iterations). Finally, the last obtained *gBest* will be the output of the algorithm that is a near-optimal solution.

3.3 Solving CRA Problem Using ACO Algorithm

In this section, a model-driven approach for solving the CRA problem using the ACO algorithm is presented. Figure 7 shows the proposed process. As other population-based meta-heuristics algorithms, the ACO algorithm consists of three main steps that will be described in future subsections.

(1) Initialization
(2) Population Generation
(3) Iterations

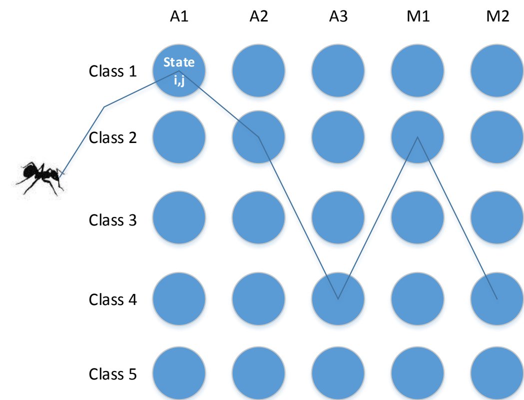


Figure 8. Modeling the Search Space for the CRA Problem.

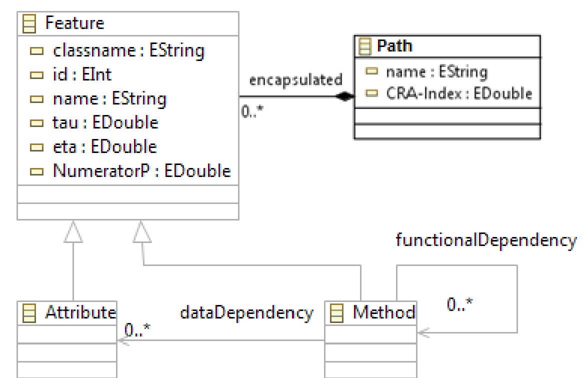


Figure 9. The Path Meta-Model.

3.3.1 Initialization

In the Initialization step, the prerequisites for the implementation of the algorithm must be provided. Among these prerequisites is the generation of a search space that should be provided as input to each ant. For this purpose, corresponding to each feature in the class model, an empty class is created, and then each feature is assigned to all the existing classes.

In the ACO algorithm, each ant should traverse a path. As shown in Figure 8, in each step, an ant selects only one state that is an assignment of a feature to a class. When the path is complete, a solution is achieved that must be evaluated.

Therefore, in the ACO algorithm, each solution model is a path traversed by an ant and conforms to a meta-model, as shown in Figure 9. Indeed the Particle (Chromosome [17]) meta-model is reused for the ACO algorithm.

In listing 8 (in appx. A.2), each feature in the input model is copied, and an empty class is created in the output model. Then, in listing 9 (in appx. A.2), each

feature is assigned to all existing classes to create a search space model that conforms to the Path meta-model.

3.3.2 Population Generation

In the population generation step, a specific number of ants are generated, and then in the Iteration step, each ant must follow a path and help the other ants to find the optimal path based on the amount of pheromone released in each iteration.

3.3.3 Iterations

In the first iteration, each ant follows a completely random path. Therefore in each step, they choose only one of the forward states and releases pheromone on it as $\tau_0 = 1$. At the end of the path, the CRA-Index of the traveled path is calculated and stores as a parameter η_{ij} on each state. This is done by transformation rules as shown in listings 10 and 11 (in appx. A.2).

In listing 10, each feature of the input model is assigned to a random class. *updateSearchSpace* transformation rules (listing 11) create a new search space in which the traversed path is copied, and other features are copied from the former search space. Thus, the pheromone value (τ) is updated in the search space for the next iteration.

In the second iteration onwards, each ant calculates the following probability (according to Equation (3)) at each step and selects the state that has the most probability value as the next state. It is necessary to mention that we involve both heuristic function and pheromone rate equally for calculating P in Equation (13) to reach the optimized solution. Therefore, the parameters are set as follows:

$$\alpha = 1 \qquad \beta = 1 \qquad \tau_0 = 1$$

$$P_{ij} = \frac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum_{j \in Root_i}(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}} \tag{13}$$

To calculate the above probability, it is necessary to place each column of Search Space (Figure 8) into a separate cluster ($Root_i$) and then calculate the probability p for all the states in each Root. This is done by *createRoletteWheel* transformation rules (listing 12 in appx. A.2). The output model conforms to the RoleteWheel meta-model (Figure 10).

Finally, based on the roulette wheel mechanism, the state with the highest p is selected as the next state along the ant's trajectory. This is done by *runAnt* transformation rules (listing 13 in appx. A.2). These rules select the feature with the highest p value from

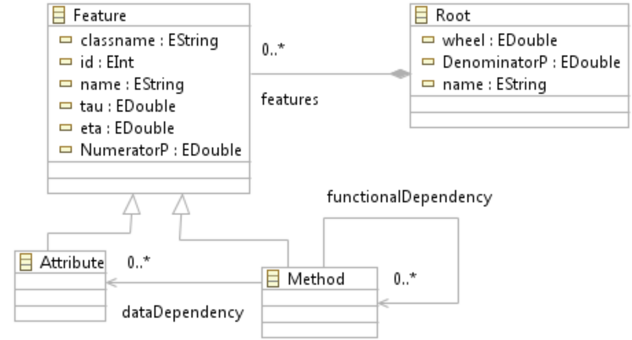


Figure 10. The RoleteWheel Meta-Model.

each Root and form a new path (solution) model. Then, the CRA-Index of the path is calculated using Java code. If this value was greater than the CRA-Index of the previous path that ant is traversed, then this value is stored as a parameter η_{ij} on the newly traversed states, and the pheromone value on each state is updated according to Equation (4). In the case of the CRA problem, Δ is calculated as follows, where Q is the coefficient of the CRA-Index effect, which is regulated in the initialization step. In this case, we set it to 2.

$$\Delta = Q * CRAIndex \tag{14}$$

Then some of that pheromone evaporates based on the evaporation rate ρ in Equation (5). Through try and error experiments, we found that it is better to set it to 0.1. Finally, the search space is updated using the *updateSearchSpace* transformation rules. If the new CRA-Index path was smaller than the previous CRA-Index path, the previous path would be selected as the ant choice.

It is necessary to mention that since the search space model is a shared space between ants, it cannot be given in parallel to all ants. Instead, each ant must affect the search space updated by the previous ants.

4 Related Work

In this section, the researches that solved the CRA problem using meta-heuristics algorithms and MDE approaches are introduced.

Faridmoayer et al. [17] presented a model-driven process based on the genetic algorithm for selecting an optimal result from the possible output models. The process is applied to the CRA case study at a high level of abstraction and is applicable to similar searched-based problems with slight changes.

Lano et al. [18] combined search-based optimization with endogenous transformation for solving the CRA problem using the UML-RSDS transforma-

tion tool. In this work, the simplification operations are first applied to the input data, and as a result, the empty classes are removed. After the refactoring phase, the genetic algorithm is applied to the data for finding the ones that are most relevant to each other and then puts them in the class. In this solution, patterns can be used for pre-processing or post-processing of data. However, the output obtained in this solution is not optimized and has a high execution time compared to other methods.

Nagy et al. [19] used the VIATRA-DSE framework and the NSGA algorithm to solve the CRA problem. VIATRA-DSE is a rule-based framework that classifies candidate output models according to different goals and criteria using graph rules and selects the best candidate. However, the results indicate that this solution is locally optimized and has low efficiency.

In [20], a solution based on the two languages of ATL and Java is developed. This method provided an automated solution using the Simulated Annealing algorithm. The ATL language is used to generate new diagrams. In this approach, the codes written in ATL and Java are very clear and understandable. However, the results are locally optimized and handling large models are not efficient and do not provide acceptable CRA-Index.

A tool called MDEOptimiser is introduced in [21] to solve the CRA problem. The framework used a graph-based transformation language called Henshin by implementing the genetic algorithm. However, the provided tool is not able to get constraints in OCL format, and there is the possibility of local optimization in this solution.

Fleck et al. [22] presented a tool, called MoMoT, to solve the CRA problem. MoMot can find optimal solutions by considering specific criteria to find models with the highest CRA-index by using the NSGAIII algorithm. MoMot is an open-source plug-in that is made of two frameworks called Henshin and MOEA (an open-source library of meta-heuristics algorithms).

John et al. [23] compare two encoding approaches for searching the optimal models systematically. The first one is a model-based encoding that represents candidate solutions as models, and the second is a rule-based encoding that represents them as a sequence of transformation rules applications. For this purpose MoMoT [22] and MDEOptimizer [24] are considered which follow the rule-based and the model-based approaches, respectively. They evaluate both approaches on a common set of optimization problems, including the CRA.

Table 1. Input Data [5].

Input Model	A	B	C	D	E
Attributes	5	10	20	40	80
Methods	4	8	15	40	80
Data Dependency	8	15	50	150	300
Functional Dependency	6	15	50	150	300

Born et al. [25] attempted to solve the CRA problem using the Henshin language and genetic algorithm. Henshin is a graph-based transformation language. Unlike other traditional optimization methods such as the hill-climbing algorithm, the genetic algorithm can implement optimization in parallel on the whole population (all models in search space). The provided solution is locally optimized with low efficiency for large-scale models.

The CRA problem is solved by Krikava [26] using the SIGMA tool. SIGMA is a domain-specific language and an API for checking model consistency and model transformation. This solution is based on a multi-objective genetic algorithm. The disadvantage of this approach is its low-efficiency to work with large-scale input models.

5 Evaluation

In this section, first, the proposed solutions are evaluated, and then the results are compared with the related works. The following criteria are considered for evaluation is this research [5]:

- Performance: By measuring execution time in different cases, the performance of the solutions could be evaluated. The time spending on reading the input and writing the output model would be disregarded.
- Optimality: This criterion is evaluated according to the CRA-Index. The higher the CRA-Index of a produced model, the higher its quality will be. This criterion can be calculated by using cohesion and coupling.

Five models with different sizes are defined with names A, B, C, D, and E. Models are in the form of a responsibility dependency graph, and their format is in XML [5]. Input models are made of some attributes, methods, and dependencies among them; however, these methods and attributes do not belong to any classes. Table 1, depicts input data in brief.

The concepts depicted in Table 1 are summarized as follows [5]:

- Attributes: Attributes are used to store concrete

data values. When used in an object-oriented language, all attributes of an object with their associated values represent the current state of the object. Changes of that state are typically executed through methods calls.
- Methods: A method is used to describe a piece of executable behavior relating to a certain functionality.
- Data Dependency: A data dependency represents a uni-directional relation from one method to an attribute, for example, a read or write access.
- Functional Dependency: A functional dependency represents a uni-directional relation from one method to another method, for example, a method call.

5.1 Experimental Setup

Since the process of implementing the optimization algorithms is random, repeatability, and reliability of results are important. Therefore, we run the algorithms over 30 times for each input model and record the best and median values as the results. This allows us to easily compare the optimality and performance of two algorithms. The generated results showed that all output models are accurate and complete. All executions are done on a system with a 3.00 GHz processor and 32 Gigabyte of RAM.

5.2 Termination Condition

The runs were continued until one of the following conditions were reached:

- The sequence of solutions converges (*i.e.*, no improvement obtained for 300 evolutions to guarantee a good level of convergence). In such a situation, we record the first time that the relevant CRA-Index is obtained as the execution time.
- A timeout termination condition was reached that we considered it on 15 minutes.

Table 2. Median and Best Results of the PSO Algorithm Over 30 Runs.

Output from	A	B	C	D	E
CRA-Index (best)	3.0	2.166	-7.875	-58.112	-155.545
CRA-Index (median)	3.0	1.1545	-10.404	-65.452	-174.405
Time in seconds (median)	44	409.099	519.504	515.5	536.5

Table 3. Median and Best Results of the ACO Algorithm Over 30 Runs.

Output from	A	B	C	D	E
CRA-Index (best)	1.0	0.279	0.768	0.435	6.856
CRA-Index (median)	-0.652	-1.053	-0.451	0.015	0.041
Time in seconds (median)	1.267	1.512	2.498	7.807	43.124

5.3 Results

Tables 2 and 3 presents the results of the evaluation of the PSO and ACO algorithms according to the CRA-Index and Execution time. From input A to input E, the number of attributes, methods, and dependencies is increasing, and also the diversity of models makes the evaluation more precise.

The results indicate that for small input models like A, PSO can produce better outputs, because PSO can create more random states. As a result, in small search spaces, peak (the best output model) can be found easier.

Moreover, the results indicate that for large models, ACO can produce better output models than PSO, because as mentioned in Section 3.3, in the ACO algorithm, each ant calculates P (Equation (13)) at each step and selects the state that has the most P value as the next state. Since the P value depends on the value of CRA-Index, in each iteration, a better path with a higher CRA-Index is achieved until the last iteration. In the PSO algorithm, as mentioned in Section 3.2, the best particle always moves randomly based on the velocity itself (generated by *randomVector* transformation module) and the other particles move towards it. Therefore the optimal position is achieved later compared to the ACO algorithms. In other words, producing solutions with PSO will converge later compared with ACO.

Figures 11 and 12 show box plots for the CRA-Index of the generated solutions for each input model. In Figure 11, for the PSO algorithm, the input models A and B are solved consistently, with a CRA-Index 3 and 1.154, respectively. While for the input model C, most of the runs return a solution with a CRA-Index around -10.404. On the other hand, for large input models like D and E, the CRA-Index is approximately about -65.452 and -174.405 individually.

Also, as shown in Figure 12, for the ACO algorithm and the input models A to C, the results are pretty varied. It can be derived from this figure that,

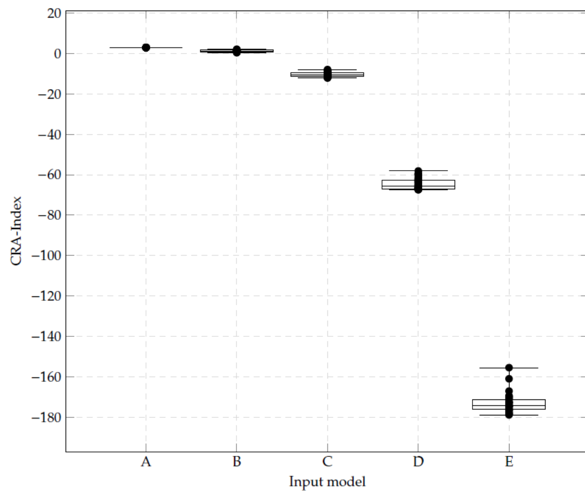


Figure 11. Values of the CRA-Index for the PSO Algorithm Over 30 Runs.

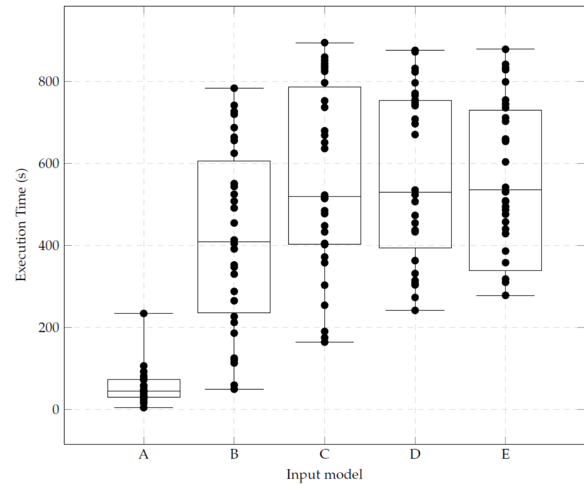


Figure 13. Execution Times of the Design Space Exploration for PSO Algorithm Over 30 Runs.

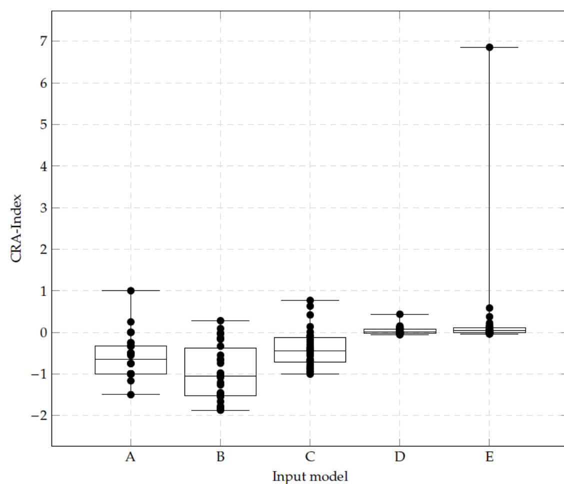


Figure 12. Values of the CRA-Index for the ACO Algorithm Over 30 Runs.

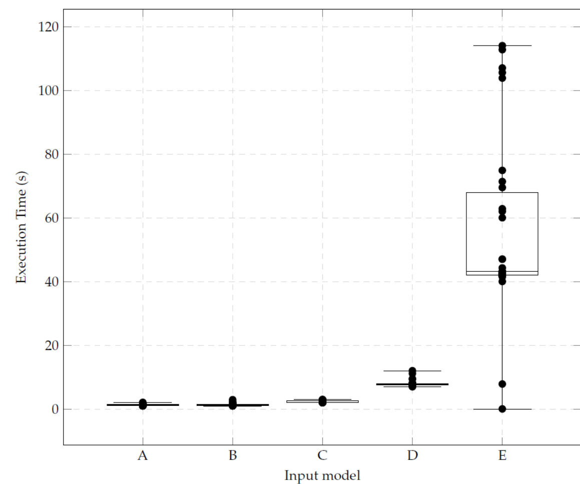


Figure 14. Execution Times of the Design Space Exploration for the ACO Algorithm Over 30 Runs.

for the ACO algorithm, CRA-Index for two input models A and B, is -0.652 and -1.053, respectively. Additionally, for input model C, CRA-Index is about -0.451. However, for the large input models D and E, the CRA-Index is 0.015 and 0.041, which are better than the results of large models in the PSO algorithm due to the mentioned features of both algorithms.

Figures 13 and 14 show execution times of the different runs in seconds. Figure 13 depicted the time execution of the PSO algorithm. Execution time greatly varies for B to E input models because, solutions that are produced by the PSO algorithm converge later than the ACO ones, as mentioned before. As a result, the second termination condition (timeout condition) is chosen. For the smallest input model, A, a solution can be found in 44 seconds, while larger input models B, C, D, E, F take 409.9, 519.50, 515.5, 536.5 and 404.91 seconds to be solved, respectively.

Also, Figure 14 shows execution times of the design space exploration for the ACO algorithm over 30 runs. It can be derived from this figure that, the result for A to D input models are pretty consistent, while it greatly varies for the largest input model, E. Execution times for A to D models are 1.26, 1.51, 2.44, and 7.80 seconds respectively, while input model E needs 43.12 seconds to be solved. The first termination condition (convergence) is chosen for the ACO algorithm, as mentioned before.

5.4 Comparing With the Related Works

In this section, the results of the proposed approaches are compared with other related works. Table 4 compares the results of provided approaches in this research with other available approaches that used GA, Simulated annealing, and ATL transformations for solving the CRA problem in the MDE community.

Table 4. Comparing the Proposed Approaches With the Related Work.

Output from			A	B	C	D	E	Average
CRA-Index	Proposed approaches	PSO	3.0	1.15	-10.40	-65.45	-174.40	-49.22
		ACO	-0.652	-1.053	-0.451	0.015	0.041	-0.42
	GA and ATL [17]		2.5	2.33	0.0	0.0	0.0	0.966
	UML-RSDS [18]		1.66	3.41	1.89	-6.34	-11.04	-2.08
	Java and ATL [20]		3.0	2.33	-6.25	-45.06	-143.24	-37.84
	MDE Optimiser	I [21]	2.33	1.67	1.40	-6.023	invalid	-
		II [21]	1.99	1.46	2.36	6.07	invalid	-
	Henshin and GA [25]		3.0	3.0	1.17	1.35	1.86	2.07
Execution Time (s)	Proposed approaches	PSO	44	409.09	519.50	515.5	536.5	404.91
		ACO	1.26	1.51	2.49	7.80	43.12	11.23
	GA and ATL [17]		3.1	1.31	1.18	1.29	1.5	1.67
	UML-RSDS [18]		16.6	0.29	33.58	115.7	571.919	147.62
	Java and ATL [20]		19.59	20.74	20.72	23.86	35.02	23.99
	MDE Optimiser	I [21]	0.55	0.89	2.45	7.42	invalid	-
		II [21]	4.18	8.45	20.58	98.24	invalid	-
	Henshin and GA [25]		3.38	3.94	4.57	8.9	24.95	9.15

We ran all approaches on a system with 32 Gigabyte of RAM and recorded the related results.

In our previous work [17], we used the Genetic algorithm to solve the CRA problem. As can be seen for small models (like A), the PSO algorithm reached the optimal solution rather than GA; and for large models (*i.e.*, D and E) ACO algorithm reached better CRA-Index than GA.

Lano [18] presented his solution in the UML-RSDS tool, which uses GA for producing optimal models. Although there are not many codes in this solution, and it has a high level of abstraction, the average CRA-Index is less than GA [17] and ACO solutions in this research.

Johnsen [20] combines Java code and ATL transformations to implement the simulated annealing algorithm. The average of the generated results for CRA-Index in this approach is also less than GA and ACO results.

6 Conclusions and Future Work

In this paper, two MDE processes using PSO and ACO algorithms were introduced to provide optimal solutions for the CRA problem. The provided solutions in this research increase the level of abstraction and finding an optimal output model in transformation specification with a large search space. The solution is compared in terms of the CRA-Indexes and execution time. The evaluation indicates that the ACO solution produces better output models than the PSO solution for large scale models. Furthermore, the proposed approaches are compared with other related works in the community, which used GA, Simulated annealing, or ATL transformations for solving the CRA problem in terms of the CRA-Indexes and execution time.

While the provided processes increase the level of abstraction, they are reusable for similar SBSE problems. The Class Modularization, EMF Refactor, Stack Load Balancing, and Class Diagram Restructuring are cases [27] that can be solved by reusing the proposed solutions in this paper considering an appropriate initial population transformation code and fitness function of the solutions. Briefly, presented solutions are based on both the MDE approach and SBSE. In MDE, the model of a software application is specified on a higher abstraction level than traditional programming languages. While the used model is on a higher abstraction level, it is much smaller compared to the same model expressed in code. In other words, each element in the model represents multiple lines of code. Hence, more function-

ality can be built at the same time. In the future, the provided algorithms are applied to more optimization cases and compared by considering more evaluation criteria. Additionally, a unified meta-model will be introduced to abstract common features of the optimization problems.

References

[1] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42 – 45, 2003. ISSN 0740-7459. doi:10.1109/MS.2003.1231150.

[2] M. Kessentini, P. Langer, and M. Wimmer. Searching models, modeling search: On the synergies of sbse and mde. In *2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE)*, pages 51–54. IEEE, 2013. ISBN 978-1-4673-6284-9. doi:10.1109/CMSBSE.2013.6604438.

[3] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):1–61, 2012. doi:10.1145/2379776.2379787.

[4] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2017.

[5] M. Fleck, J. Troya, and M. Wimmer. The class responsibility assignment case. *TTC 2016: 9th Transformation Tool Contest, co-located with the 2016 Software Technologies: Applications and Foundations (STAF 2016)(2016), p 1-8*, pages 1–8, 2016. ISSN 1613-0073.

[6] M. Kessentini, H. Sahraoui, M. Boukadoum, and O. Omar. Search-based model transformation by example. *Software & Systems Modeling*, 11 (2):209–226, 2010. doi:10.1007/s10270-010-0175-7.

[7] E. Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.

[8] L. Bianchi, M. Dorigo, L. Gambardella, and W. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8(2):239–287, 2009. doi:10.1007/s11047-008-9098-4.

[9] Martin Fleck, Javier Troya, and Manuel Wimmer. Marrying search-based optimization and model transformation technology. In *1st North American Search Based Software Engineering Symposium (NasBASE 2015)*, pages 1–16, 2015.

[10] Eberhart and Y. Shi. Particle swarm optimization: developments, applications and resources. In *Proceedings of the 2001 Congress on evolutionary computation*, pages 81–86. IEEE, 2001. ISBN 0-7803-6657-3. doi:10.1109/CEC.2001.934374.

[11] Wikimedia. https://upload.wikimedia.org/wikipedia/commons/thumb/a/af/Aco_branches.svg/2000px-Aco_branches.svg.png, Date Accessed: November 7, 2019.

[12] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66, 1997. ISSN 1089-778X. doi:10.1109/4235.585892.

[13] S. Goss, S. Aron, J. Deneubourg, and J. M. Pasteels. Self-organized shortcuts in the argentine ant. *Naturwissenschaften*, 76(12):579–581, 1989. doi:10.1007/BF00462870.

[14] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. doi:10.1016/j.entcs.2005.10.021.

[15] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008. doi:10.1016/j.scico.2007.08.002.

[16] James Kennedy. Particle swarm optimization. In *Encyclopedia of machine learning*, pages 760–766. Springer, 2011.

[17] S. Faridmoayer, M. Sharbaf, and S. Kolahdouz-Rahimi. Optimization of model transformation output using genetic algorithm. In *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pages 0203–0209. IEEE, 2017. ISBN 978-1-5386-2641-2. doi:10.1109/KBEI.2017.8324973.

[18] K. Lano, S. Kolahdouz-Rahimi, and S. Yassipour-Tehrani. Solving the class responsibility assignment case with uml-rsds. In *TTC@STAF*, pages 9–14, 2016.

[19] A. Nagy and G. Szárnyas. Class responsibility assignment case: a viatra-dse solution. In *TTC@STAF*, 2016.

[20] L. Johnsen Arne, F. Macıas, and A. Rutle. Solving the class responsibility assignment using java and atl. *TTC@STAF*, pages 20–23, 2016.

[21] A. Burdusel and S. Zschaler. Model optimisation for feature class allocation using mdeoptimiser: A ttc 2016 submission. In *TTC@STAF*, pages 33–38, 2016.

[22] R. Bill, M. Fleck, J. Troya, T. Mayerhofer, and M. Wimmer. A local and global tour on momot. *Software & Systems Modeling*, 18(2):1017–1046, 2019. doi:10.1007/s10270-017-0644-3.

[23] S. John, A. Burdusel, R. Bill, D. Strüber, G. Taentzer, S. Zschaler, and M. Wimmer. Searching for optimal models: Comparing two encoding approaches. *Software & Systems Modeling*, 18(2):1017–1046, 2019. ISSN 978-3-88579-694-7. doi:10.18420/SE2020_30.

[24] A. Burdusel, S. Zschaler, and D. Strüber.

Mdeoptimiser: a search based model engineering tool. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 12–16. ACM, 2018. doi:10.1145/3270112.3270130.

[25] K. Born, S. Schulz, D. Strüber, and S. John. Solving the class responsibility assignment case with henshin and a genetic algorithm. In *TTC@STAF*, pages 45–54, 2016.

[26] F. Krikava. Solving the ttc'16 class responsibility assignment case study with sigma and multi-objective genetic algorithms. In *TTC@STAF*, pages 55–60, 2016.

[27] M. Fleck, J. Troya, and M. Wimmer. Search-based model transformations. *Journal of Software: Evolution and Process*, 28(12):1081–1117, 2016. doi:10.1002/smr.1804.

A Details of Our Solution

A.1 PSO Algorithm Transformation Rules

Listing 5: *randomPop* Transformation Rules.

```
-- @atlcompiler emftvm
module class2particle;
create OUT: particleCRA from IN: architectureCRA ;

helper def : id : Integer = 0;

helper def : fSize : Integer = architectureCRA!Feature.
     allInstances().size() ;

helper def : testRand(i : Integer) : Integer =
  let rnd : "#native"!"java::util::Random" = "#native"!"
       java::util::Random".newInstance() in rnd.nextInt
       (i) + 1;

rule att2att{
  from
    attr : architectureCRA!Attribute
  to
    parA : particleCRA!Attribute(
      name <- attr.name,
      classname <- 'class' + thisModule.testRand(
           thisModule.fSize).toString()
    )
  do{
     thisModule.id <- thisModule.id + 1;
     parA.id <- thisModule.id;
   }
}

rule method2method{
  from
    cm : architectureCRA!Method
  to
    parM : particleCRA!Method(
      name <- cm.name,
      functionalDependency <- cm.functionalDependency,
      dataDependency <- cm.dataDependency,
      classname <- 'class' + thisModule.testRand(
           thisModule.fSize).toString()
    )
  do{
     thisModule.id <- thisModule.id + 1;
     parM.id <- thisModule.id;
   }
}
```

Listing 6: Concatenating Two Vectors.

```
-- @atlcompiler emftvm
module UpdateVector;
create OUT : velocityVector from IN1 : velocityVector,
     IN2 : velocityVector;

rule CopyVector1{
  from
     ps1: velocityVector!Vector in IN1
  to
     ou: velocityVector!Vector in OUT(
        has <- ps1.has
     )
```

```
}

rule CopySourcePoint1{
  from
    ps1: velocityVector!Point in IN1
  to
    ou : velocityVector!Point in OUT(
      X <- ps1.X,
      Y <- ps1.Y
    )
}

rule CopyVector2{
  from
    ps2: velocityVector!Vector in IN2
  to
     ou2: velocityVector!Vector in OUT(
        has <- ps2.has
     )
}

rule CopySourcePoint2{
  from
    ps2: velocityVector!Point in IN2
  to
    ou2 : velocityVector!Point in OUT(
      X <- ps2.X,
      Y <- ps2.Y
    )
 }
```

Listing 7: Updating Particle Position.

```
-- @atlcompiler emftvm
module UpdateParticle;
create OUT : particleCRA from IN1 : particleCRA, IN2 :
     velocityVector;

helper context particleCRA!Feature def : getid() :
     Integer = self.id;

helper context particleCRA!Feature def : getClassname()
      : String = self.classname;

helper context particleCRA!Feature def : newClassname()
      : String =
  if velocityVector!Point.allInstances()->exists(c |
       self.getid() = c.X)
  then
    'class' + velocityVector!Point.allInstances()->
         select(c | self.getid() = c.X)->last().Y.
         toString()
  else
    self.getClassname()
  endif;

rule ExchangeAtt{
  from
    att : particleCRA!Attribute in IN1
    to
     ou : particleCRA!Attribute in OUT(
        name<-att.name,
      id <- att.id,
      classname <-att.newClassname()
  )
}
```

```
rule ExchangeMethod{
  from
    me : particleCRA!Method in IN1
    to
    ou : particleCRA!Method in OUT (
        name<-me.name,
      id <- me.id,
      classname <- me.newClassname(),
      dataDependency <- me.dataDependency,
      functionalDependency <- me.functionalDependency
    )
}
```

A.2 ACO Algorithm Transformation Rules

Listing 8: *CompleteClassModel* Transformation Rules.

```
-- @atlcompiler emftvm
module CompleteClassModel;
create OUT : architectureCRA1 from IN : architectureCRA;

helper def : id : Integer = 0;

rule att2att{
  from
    ft : architectureCRA!Attribute
  to
    ft1 : architectureCRA1!Attribute(
      name <- ft.name
    ),
    class : architectureCRA1!Class
    do{
      thisModule.id <- thisModule.id + 1;
      class.name <- 'class' + thisModule.id.toString();
    }
}

rule method2method{
  from
    cm : architectureCRA!Method
  to
    cm1 : architectureCRA1!Method(
      name <- cm.name,
      functionalDependency <- cm.functionalDependency,
      dataDependency <- cm.dataDependency
    ),
    class : architectureCRA1!Class
    do{
      thisModule.id <- thisModule.id + 1;
      class.name <- 'class' + thisModule.id.toString();
    }
}
```

Listing 9: *SearchSpace* Transformation Rules.

```
-- @atlcompiler emftvm

module searchSpace;
create OUT: pathCRA from IN: architectureCRA ;

helper def : id : Integer = 0;

rule att2att{
  from
    ft : architectureCRA!Attribute , class:
```

```
        architectureCRA!Class
  to
    State : pathCRA!Attribute(
      name <- ft.name,
      classname <- class.name
    )
  do{
    thisModule.id <- thisModule.id + 1;
    State.id <- thisModule.id;
  }
}

rule method2method{
  from
    cm : architectureCRA!Method , class: architectureCRA
        !Class
  to
    State : pathCRA!Method(
      name <- cm.name,
      classname <- class.name
    )
  do{
    thisModule.id <- thisModule.id + 1;
    State.id <- thisModule.id;
  }
}
```

Listing 10: *createApath* Transformation Rules.

```
-- @atlcompiler emftvm
module createApath;
create OUT: pathCRA from IN: architectureCRA ;

helper def : id : Integer = 0;

helper def : fSize : Integer = architectureCRA!Feature.
     allInstances().size() ;

helper def : testRand(i : Integer) : Integer =
  let rnd : "#native"!"java::util::Random" = "#native"!"
       java::util::Random".newInstance() in rnd.nextInt
       (i) + 1;

rule att2att{
  from
    ft : architectureCRA!Attribute
  to
    ch : pathCRA!Attribute(
    name <- ft.name,
    classname <- 'class' + thisModule.testRand(
         thisModule.fSize).toString()
  )
  do{
    thisModule.id <- thisModule.id + 1;
    ch.id <- thisModule.id;
  }
}
rule method2method{
  from
    cm : architectureCRA!Method
  to
    ch : pathCRA!Method(
    name <- cm.name,
    functionalDependency <- cm.functionalDependency,
    dataDependency <- cm.dataDependency,
    classname <- 'class' + thisModule.testRand(
         thisModule.fSize).toString()
  )
  do{
    thisModule.id <- thisModule.id + 1;
    ch.id <- thisModule.id;
  }
}
```

Listing 11: *updateSearchSpace* Transformation Rules.

```
-- @atlcompiler emftvm

module updateSearchSpace;
create OUT: pathCRA from IN1: pathCRA , IN2: pathCRA;

-- IN1 is searchSpace, IN2 is the Path, OUT is the
     updated searchSpace

rule updateAttr{
  from
    ft1 : pathCRA!Attribute in IN1 , ft2 : pathCRA!
         Attribute in IN2
    (ft1.name=ft2.name and ft1.classname=ft2.classname)
  to
    State : pathCRA!Attribute(
      name <- ft2.name,
      classname <- ft2.classname,
      id <- ft2.id,
      tau <- ft2.tau, -- tau will be updated
      eta <- ft2.eta, -- eta will be updated
      NumeratorP <- ft2.NumeratorP
    )
}

rule updateMethod{
  from
    cm1 : pathCRA!Method in IN1 , cm2 : pathCRA!Method in
          IN2
    (cm1.name=cm2.name and cm1.classname=cm2.classname)
  to
    State : pathCRA!Method(
      name <- cm2.name,
      classname <- cm2.classname,
      id <- cm2.id,
      tau <- cm2.tau, -- tau will be updated
      eta <- cm2.eta, -- eta will be updated
      NumeratorP <- cm2.NumeratorP
    )
}

rule att2att{
  from
    ft1 : pathCRA!Attribute in IN1 , ft2 : pathCRA!
         Attribute in IN2
    (ft1.name=ft2.name and ft1.classname<>ft2.classname
         )
  to
    State : pathCRA!Attribute(
      name <- ft1.name,
      classname <- ft1.classname,
      id <- ft1.id,
      tau <- ft1.tau,
      eta <- ft1.eta,
      NumeratorP <- ft1.NumeratorP
    )
}
```

```
rule method2method{
  from
    cm1 : pathCRA!Method in IN1 , cm2 : pathCRA!Method in
          IN2
     (cm1.name=cm2.name and cm1.classname<>cm2.
          classname)
  to
    State : pathCRA!Method(
      name <- cm1.name,
      classname <- cm1.classname,
      id <- cm1.id,
      tau <- cm1.tau,
      eta <- cm1.eta,
      NumeratorP <- cm1.NumeratorP
    )
}
```

Listing 12: *createRoletteWheel* Transformation Rules.

```
-- @atlcompiler emftvm
module createRoleteWheel;
create OUT: RoleteWheel from IN1: pathCRA1 , IN2:
     pathCRA2;

-- IN1 is a random path, IN2 is the searchSpace

helper def : createwheel(numerators: Sequence(Real), d:
      Real) : Sequence(Real) =
  numerators->iterate(e; p: Sequence(Real) = Sequence{}
        | p->append(e/d) );

helper def : getResources(f: pathCRA1!Feature) :
     Sequence(pathCRA2!Feature) =
  let a: Sequence(pathCRA2!Feature) = pathCRA2!Feature.
       allInstances()->select(e | e.name = f.name) in
  a->excluding(a->last());

helper def : shift(numerators: Sequence(Real)) :
     Sequence(Real) =
  let min : Real = numerators->iterate(e; min: Real =
       1000 | e.min(min) ) in
  if (min < 0 ) then
    numerators->iterate(e; shift : Sequence(Real) =
         Sequence{} | shift->append( e + min.abs() ) )
  else
    numerators
  endif;

helper def : getNumerators(f: Sequence(pathCRA2!Feature
     )) : Sequence(Real) =
  let numerators: Sequence(Real) = f->iterate(e; nums:
       Sequence(Real) = Sequence{} | nums->append(e.
       NumeratorP) ) in
  thisModule.shift(numerators);

helper def : SumOfNumerators(res : Sequence(pathCRA2!
     Feature) ) : Real =
  thisModule.getNumerators(res)->iterate(e; sum : Real
       = 0.0 | sum + e);

rule createRoleteWheel{
  from
    ft: pathCRA1!Feature in IN1

    using {
      res : Sequence(pathCRA2!Feature) = thisModule.
           getResources(ft);
      denominator : Real = thisModule.SumOfNumerators(
           res);
    }
  to
    ch: RoleteWheel!Root(
      name <- ft.name,
      features <- res,
      wheel <- thisModule.createwheel(thisModule.
           getNumerators(res), denominator),
      DenominatorP <- denominator
    )
}

rule attr2attr{
  from
    ft : pathCRA2!Attribute in IN2
  to
    r : RoleteWheel!Attribute(
      name <- ft.name,
      id <- ft.id,
      classname <- ft.classname,
      tau <- ft.tau,
      eta <- ft.eta,
      NumeratorP <- ft.NumeratorP
    )
}

rule method2method{
  from
    ft : pathCRA2!Method in IN2
  to
    r : RoleteWheel!Method(
      name <- ft.name,
      id <- ft.id,
      classname <- ft.classname,
      tau <- ft.tau,
      eta <- ft.eta,
      NumeratorP <- ft.NumeratorP
    )
}
```

Listing 13: *runAnts* Transformation Rules.

```
-- @atlcompiler emftvm
module runAnts;
create OUT: pathCRA from IN: RoleteWheel, IN1: pathCRA1;

-- IN1 is popModel,
-- OUT is a random path

helper def : rangeMax( c: Sequence(Real) ) : Real = c.at(
     c.size());

helper def : createdRanNum( i : Real ) : Real =
  let rnd : "#native"!"java::util::Random" = "#native"!"
       java::util::Random".newInstance() in
  i*rnd.nextDouble();

helper def : RoleteWhileSelection( c: Sequence(Real) ) :
      Integer =
  let max : Real = c->iterate(e; max: Real = c->first() |
       e.max(max) ) in
  c->indexOf( max );
```

```
rule selectAPath{
  from
    f : pathCRA1!Attribute
  using {
      root : RoleteWheel!Root = RoleteWheel!Root.
           allInstances()->select(r | r.name = f.name)->
           first();
      index : Integer = thisModule.RoleteWhileSelection(
           root.wheel);
      ft: RoleteWheel!Feature = root.features.at(index);

    }

  to
    ch: pathCRA!Attribute(  -- The RoleteWhileSelection
          should be called only once! for each root
      name <- ft.name,
      id <- ft.id,
      classname <- ft.classname,
      tau <- ft.tau,
      eta <- ft.eta,
      NumeratorP <- ft.NumeratorP
    )
}

rule selectAPath2{
  from
    cm : pathCRA1!Method
  using {
      root : RoleteWheel!Root = RoleteWheel!Root.
           allInstances()->select(r | r.name = cm.name)
           ->first();
      index : Integer = thisModule.RoleteWhileSelection(
           root.wheel) ;
      ft: RoleteWheel!Feature = root.features.at(index);

    }
  to
    ch: pathCRA!Method(
      name <- ft.name,
      id <- ft.id,
      classname <- ft.classname,
      tau <- ft.tau,
      eta <- ft.eta,
      NumeratorP <- ft.NumeratorP,
      functionalDependency <- cm.functionalDependency,
      dataDependency <- cm.dataDependency
    )
}
```

Sogol Faridmoayer is a Ph.D. student at University of Montreal, Montreal, QC, Canada. She received her B.Sc. in computer software engineering from the University of Isfahan in May 2015. Then she studied at the M.Sc. degree in Software Engineering at the same University and graduated in September 2018. Her interests include Model-Driven Software Engineering, ModelTransformation, Search-Based Software Engineering. She is now a member of the GEODES Software Engineering Research Group at the University of Montreal.

Samaneh HoseinDoost is a Ph.D. Candidate at the University of Isfahan. She received her B.Sc. in computer software engineering from the University of Isfahan in September 2014. Then she studied at the M.Sc. degree in Software Engineering at the same University and graduated in April 2017. Her interests include Model Driven Software Engineering, Multi-Agent Systems, and crisis management systems. She is a member of the Model-Driven Software Engineering Research Group (MDSERG) at the University of Isfahan.

Shekoufeh Kolahdouz-Rahimi is an Assistant Professor in the Computer Engineering Department at the University of Isfahan. She is an active member of the Model Driven Software Engineering Research Group at this University. She has completed her Ph.D. in Computer Science at Kings College London in 2013. Her research interest includes Design patterns for Model Transformation, Specification and Verification of Model Transformations, Bidirectional Model Transformations, Domain-Specific Modelling Languages, and Search-Based Software Engineering.

Bahman Zamani received his B.Sc. from the University of Isfahan, Isfahan, Iran, in 1991, and his M.Sc. from the Sharif University of Technology, Tehran, Iran, in 1997, both in Computer Engineering (Software). He obtained his Ph.D. in Computer Science from Concordia University, Montreal, QC, Canada, in 2009. From 1998 to 2003, he was a researcher and faculty member of the Iranian Research Organization for Science and Technology (IROST)—Isfahan branch. Dr. Zamani joined the Faculty of Computer Engineering at the University of Isfahan in 2009, as an Assistant Professor. In 2018, he was promoted to Associate Professor. His main research interest is Model-Driven Software Engineering (MDSE).