Received July 7, 2020, accepted August 7, 2020, date of publication August 11, 2020, date of current version August 31, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3015864

Meta-Heuristics for Solving the Software Component Allocation Problem

ISSAM AL-AZZONI AND SAQIB IQBAL
Department of Software Engineering and Computer Science, Al Ain University, Al Ain 64141, United Arab Emirates
Corresponding author: Issam Al-Azzoni (issam.alazzoni@aau.ac.ae)

ABSTRACT The software component allocation problem is concerned with mapping a set of software components to the computational units available in a heterogeneous computing system while maximizing a certain objective function. This problem is important in the domain of component-based software engineering, and solving it is not a trivial task. In this paper, we demonstrate a software framework for defining and solving component allocation problem instances. In addition, we implement two meta-heuristics for solving the problem. The experiments show that these meta-heuristics achieve good performance. The framework is designed to be extensible and therefore other researchers can conveniently use it to implement new meta-heuristics for solving the software component allocation problem.

INDEX TERMS Component allocation, model-driven engineering, embedded systems, heterogeneous systems, genetic algorithms, ant colony optimization.

I. INTRODUCTION

Computer systems today have become more heterogeneous than ever before, with systems consisting of different types of processors and computing resources. Furthermore, advances in component-based software engineering have enabled designers and architects with the freedom to make different allocations of software components on the available computational units. In the domain of component-based software engineering, a software component is a unit of composition with contractually defined interfaces which can be deployed independently without concerns of timing conflicts and precedence [1]–[3]. However, the heterogeneity of systems presents a big challenge to designers and architects. While several allocations can be functionally correct, one or more of these allocations can have better quality aspect(s) than the remaining allocations. The component allocation problem is the problem of finding the allocation that is functionally correct (*i.e.,* does not violate any specified constraint) and that maximizes a certain quality metric [4]–[6]. Solving the component allocation problem is not trivial, and several previous research projects have looked into this problem [7]–[9]. New tools and frameworks are needed to help in making such important decisions.

The component allocation problem is an NP-hard problem [6], [10], and thus the difficulty of solving it increases exponentially as the numbers of components and computational units increase. Therefore, using traditional greedy search methods can be ineffective in solving the component allocation problem. To overcome this challenge, meta-heuristics have been used to solve the component allocation problem. These meta-heuristics usually succeed in finding optimal or good sub-optimal solutions in very short time. This is especially true in cases where an exhaustive search strategy becomes infeasible due to the large size of the search space.

This paper presents a new extensible framework for modeling and solving the component allocation problem. The framework is implemented as an Eclipse Java project utilizing the Eclipse Modeling Framework (EMF) [11]. The core of the framework is a new meta-model for the component allocation problem. Test models defining component allocation problem instances must conform to this meta-metamodel. We have created several of these test models which are discussed in Section V. Interested authors can use these test models when validating the performance of new meta-heuristics for solving the component allocation problem. New test models can also be added in the future to the framework. By creating this framework, approaches and advances in the domain of Model-Driven Engineering (MDE) can be utilized in the domain of software component allocation. In addition, we have implemented two meta-heuristics within the framework: the first one is a Genetic Algorithm (GA) and the second is an Ant Colony Optimization (ACO) algorithm. Although GAs have been used to solve the component allocation problem before, this paper is the first one to apply an ACO algorithm for solving the component allocation problem with a single objective function. In this paper, a detailed comparison study between

The associate editor coordinating the review of this manuscript and approving it for publication was Sotirios Goudos.

the two meta-heuristics is also presented. The framework is built with extensibility in mind, and interested authors can use it, with little effort, to implement new meta-heuristics and to analyze their performance using the same set of test models.

The main contributions of this paper are the following:

1) An extensible framework for solving the component allocation problem. The framework is based on a meta-model defining the component allocation problem. The framework can be extended with new solution methods for solving the component allocation problem.
2) Implementation of an ACO algorithm for solving the component allocation problem. This paper is the first one to propose using an ACO algorithm for solving the single-objective component allocation problem. ACO outperforms GA-based implementations on several evaluation cases as presented in Section V.

The remainder of this paper is organized as follows: Section II discusses related work. Section III defines the component allocation problem, while Section IV presents several meta-heuristic algorithms for solving the problem. Section V evaluates the implementations of these algorithms. Finally, Section VI concludes the paper with outlines for future research.

II. RELATED WORK

Švogor *et al.* [12] define a software component allocation model for heterogeneous systems. In addition, the authors implement a method for finding optimal allocations using GA. Analytic Hierarchy Process (AHP) [13] is used to assign weights to the different resource consumption costs. Švogor and Carlson [5] present SCALL (Software Component Allocator) which is a prototype Eclipse plug-in, based on Eclipse Modeling Framework (EMF) [14] and Graphical Modeling Project (GMF) [15], used to create models of allocation problems on heterogeneous systems and solve those allocation problems. A GA implemented in Python is used to find sub-optimal allocations. In Švogor *et al.* [6], the authors present a comprehensive framework, called SCAF (Software Component Allocation Framework). The framework encompasses a theoretical model for component allocation on heterogeneous systems. The authors validate SCAF on a real-world system demonstration. In addition, the authors implement SCAF as part of the SCALL plug-in and add a simulated annealing-based solver. Our component allocation model is based on the model defined by Švogor *et al.* [6], [12]. With respect to the SCALL plug-in, it does not provide direct methods for extending solution methods to the component allocation problem. On the other hand, our tool has been designed to be extensible in such a way that it supports adding new solution methods.

A generic and extensible framework, called Deployment Improvement Framework (DIF), for optimizing deployment architectures specified in AADL [16] is presented by Malek *et al.* [7]. The framework gives a formal definition of the component allocation problem and implements several algorithms for solving the problem. The authors present a tool to help users in applying the framework.

Feljan *et al.* [17] present a method for allocating software tasks to the cores of a multicore system. The method is designed for soft real-time embedded multicore systems in which timing is critical to the correctness of the system but few deadline misses are tolerated. The method is based on an iterative model-driven optimization cycle employing simulation guided local search. Two main performance metrics are considered to compare task allocations: timeliness (the average number of missed deadlines) and the computational load distribution.

Wang *et al.* [9] present a method for component allocation with multiple resource constraints for large embedded real-time systems. The method uses an informed branch-and-bound and forward checking mechanism. The method was implemented in the Automatic Integration of Reusable Embedded Software (AIRES) toolkit. Given a model of software components with resource consumptions and a model of a target platform's set of devices with resource constraints, the method aims to group components into partitions such that the resource constraints are respected and the amount of communication among the partitions is within the available link capacity. The method does not have an objective function for optimization, but only aims to find a feasible partition in a scalable way.

A component allocation method for series-parallel systems with interchangeable elements is proposed by Yamachi *et al.* [18]. The method applies a Multi-Objective Genetic Algorithm (MOGA) to obtain the Pareto solutions with two main objective functions: system cost and reliability. In the allocation model considered in the method, components can fail. Components can have varying reliability and cost attributes. In the context of series-parallel systems, parallel systems are serially arranged. This enables redundancy allocation in which several components can be allocated on the same subsystem in order to increase system reliability. The method aims to allocate the components in order to achieve maximum system reliability and minimum cost.

Pohlmann and Hüwe [8] propose a model-driven approach for specifying allocation problems and automatically computing feasible allocations. The authors propose a domain-specific language named ASL and an Eclipse-based tool that an allocation engineer can use to specify the allocation constraints. The allocation problem is formulated as a 0-1 Integer Linear Program (ILP). Model transformation is used to transform a model of the component allocation problem specified in ASL into a form that can be processed by an ILP-solver. The approach was validated in the context of an automotive case study. The approach only finds feasible allocations (no optimal solutions) and may not scale to large problems since it is based on exact-ILP solvers. The approach can cover several types of allocation constraints such as timing, priority, deadlines, and schedulability.

Koziolek *et al.* [19] present an approach called PerOpteryx that uses a metaheuristic search guided by architectural tactics to improve software architectural models. The architectural models are specified with the Palladio Component

Model (PCM) [20]. For performance analysis, the approach transforms a PCM model into a layered queueing network (LQN) [21]. The approach is designed to find Pareto-optimal solutions with respect to two objectives: response time and server costs. The approach uses the multi-objective evolutionary algorithm NSGA-II [22] integrated with design-level architectural tactics.

Li *et al.* [23] introduce a toolkit, called AQOSA (Automated Quality-driven Optimization of Software Architecture) for component-based software design. AQOSA applies several evolutionary multi-objective optimization algorithms to find approximations to the Pareto optimal sets. The optimization process in AQOSA starts with an initial input software architecture modeled in a supported software architecture modeling language such as AADL. AQOSA then iteratively generates alternative architecture models through genetic operations on a genotype representation which allows several degrees of freedom for exploration, including component allocation. Three objectives are considered: processor utilization, cost, and data flow latency. Simulation, using ADeS [24], is used to evaluate and compare the solutions.

Aleti *et al.* [25] present an extendable tool, called ArcheOpterix, which provides a framework for architecture optimization for AADL specifications. The tool provides a platform for users to implement architecture optimization algorithms. Since ArcheOpterix only accepts AADL models, it depends on AADL's development environment OSATE [26]. In Aleti *et al.* [27], ArcheOpterix is used to compare the performance of Pareto-Ant Colony Optimization (P-ACO) with the Multi-Objective Genetic Algorithm (MOGA) on component deployment optimization problems. Two objectives are optimized: the data transmission reliability and the communication overhead. In this context, the deployment problem is modeled as a bi-objective optimization problem. Based on the validation experiments, the authors observe that P-ACO performs similar to MOGA, however P-ACO's optimization progress stagnates after a certain number of iterations.

Ashraf *et al.* [28] present a multi-objective Ant Colony System (ACS) algorithm for cloud-based software component deployment problems. Three objective functions are considered: cost, performance, and reliability. The algorithm aims to find a set of Pareto-optimal deployment configurations. The algorithm exploits three generic architectural Degrees of Freedom (DoFs) to create architecture alternatives: component allocation, virtual machine (VM) selection, and the number of VMs. The algorithm is implemented in Java, and compared with the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [22].

The work of Campeanu *et al.* [29], [30] provides an optimization model for allocating components in heterogeneous CPU-GPU Embedded Systems. The optimization model is defined as a mixed integer nonlinear programming problem and solved using CPLEX [31]. The work assumes the use of Rubus [32] as a component model for the embedded system being optimized. In addition, the platform considered contains a single CPU-GPU chip. Campeanu and Saadatmand [33] propose a run-time component allocation method in CPU-GPU Embedded Systems. The proposed method dynamically assigns components over hardware. The allocation model is also defined as a mixed-integer nonlinear programming (MINLP) model.

III. PROBLEM STATEMENT

Consider an embedded system consisting of m computational units. Each computational unit offers a number of resources l, *e.g.* computation, memory, and energy resources. There are n software components that need to deployed on the computational units. Each software component can only be allocated on a single computational unit.

The component allocation problem is to find an optimal allocation (mapping) of the components to the computational units. An allocation can be represented as a permutation with repetition $(p_1, \ldots, p_n)$ which assigns every component i to a computational unit p_i. A feasible allocation is an allocation such that the resources consumed by the components do not exceed the resource capacities that the computational units provide. In addition, a feasible allocation must not violated any architectural constraint. These concepts will be introduced shortly. An optimal allocation is a feasible allocation such that the objective (cost) function is minimized.

In order for an allocation to be feasible, it has to satisfy the following condition:

$$\sum_{i,p_i=j} (t_{ip_ik}) \leq r_{jk} \tag{1}$$

for any computational unit j and for all resources k. Table 1 lists the definitions of several matrices characterizing the component allocation problem. The elements t_{ijk} and r_{jk} which appear in Equation 1 are defined in the table. In addition, a feasible allocation must not violate any architectural constraints. For example, a co-allocation constraint requires that a component must be allocated on a particular computational unit. On the other hand, an anti-allocation constraint requires that a component must never allocated on a particular computational unit.

The cost of an allocation $(p_1, \cdots, p_n)$ can be computed using the following cost function:

$$w = \sum_{k=1}^{l} f_k \sum_{i=1}^{n} t_{ip_ik} \tag{2}$$

Here, f_k is a trade-off factor whose purpose is to specify the weight of the corresponding resource k in the cost function.

The component allocation problem is a kind of a quadratic assignment problem which is an NP-hard problem [10]. Therefore, the search state space grows exponentially with the problem size. Hence, for very large problem sizes, it is necessary to use meta-heuristics which find good approximations to the optimal allocations. In this paper, we present two such meta-heuristics (see Section IV).

The meta-model defining an allocation problem is depicted in Figure 1. An *AllocationProblem* is composed of the

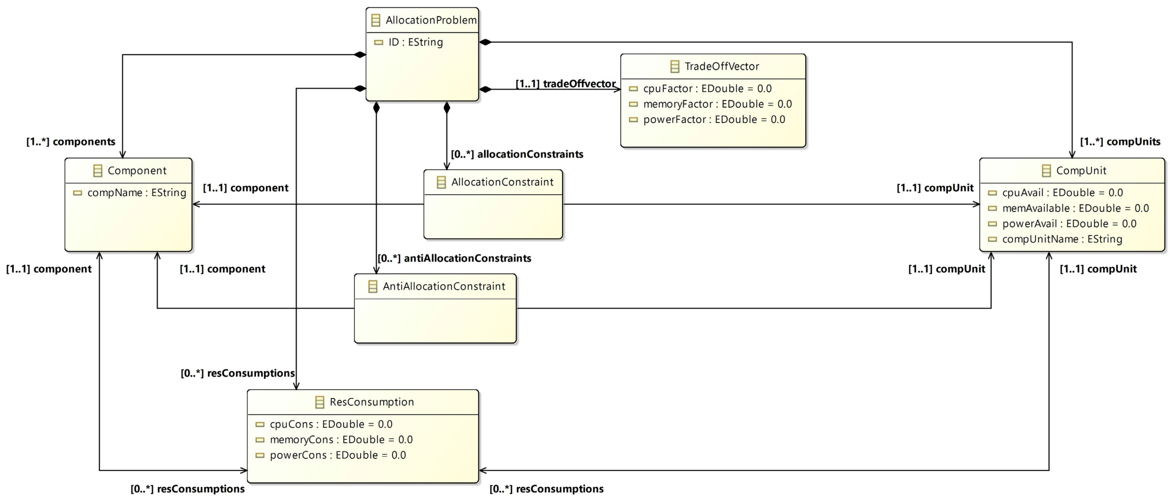


FIGURE 1. The Component Allocation Problem (*CAP*) meta-model.

TABLE 1. Component Allocation Problem Input Matrices.

Matrix or Vector	Symbol	Dimension	Description
The Component Resource Consumption	T	$n \times m \times l$	The element t_{ijk} is the amount of the kth resource required by the ith component when allocated on the jth computational unit
The Computational Unit Resource Capacity	R	$m \times l$	The element r_{jk} is the amount of the kth resource capacity of the jth computational unit
The Trade-off Vector	F	$1 \times l$	The element F_l is the trade-off weight assigned to the lth resource

following definitions. First, the *Component* class represents the components. Each component has a name. Second, the *CompUnit* class represents the computational units. Each *CompUnit* has a name in addition to its available CPU, memory, and power resource capacities. Third, the *TradeOffVector* represents the trade-off vector. Fourth, the *ResConsumption* class represents the component resource consumption matrix. Each *ResConsumption* element shows the CPU, memory, and power resource consumptions of the associated *Component* on the associated *CompUnit*. Finally, the architectural constraints are represented using the *AllocationConstraint* and *AntiAllocationConstraint* classes. For example, an *AllocationConstraint* element linking a *Component* to a *CompUnit* represents the architectural constraint that this component must be allocated on that particular computational unit.

We developed a complete Java project which can read models conforming to the *CAP* meta-model in Figure 1. Then, it calls implemented solvers that attempt to find the exact optimal cost (in the case of exact solution methods) or the best approximation to the optimal cost (in the case of meta-heuristics). The project files are available on GitHub at https://github.com/ialazzon/ComponentAllocationProblem. Several test models are also included.

The project includes solvers based on the Genetic Algorithm and Ant Colony Optimization. These are presented and evaluated in the subsequent sections of the paper. Furthermore, the project was designed with extensibility in mind. Interested authors can add new solvers, and compare their performance with already implemented solvers on the same component allocation problem models. To add a new solver, the following method must be implemented:

```
  public static double solveAllocation
Problem (AllocationProblemCon allocation)
```

This requires importing the class `allocationProblem.AllocationProblemCon`. This class encapsulates the component allocation problem; it provides access to the values of the component allocation problem parameters which are read from the input models: `m`, `n`, `R`, `Tcpu`, `Tmemory`, `Tpower`, and `F`. In addition, it implements the methods: `isValid(int allocation[])` which returns true if the allocation represented by the argument is true and false otherwise, and `cost(int`

TABLE 2. GA Settings.

Generations	**100**
Mutation Rate	**0.05**
Crossover Rate	**0.95**
Population Size	**80**
Selection Algorithm	**Tournament**
Tournament Size	**5**
Elitism Count	**2**

`allocation[])` which returns the cost of the allocation. Note that a reference to an array of integers is used to represent an allocation. Each element in the array represents the assignment of the component corresponding to the element's index to one of the computational units. For example, `allocation[i] = j` denotes the assignment of component i to computational unit j.

IV. APPROACH

In this section, we present the implementations of two meta-heuristics for solving the component allocation problem. The first one is based on GA, while the second one is based on ACO.

A. GENETIC ALGORITHM IMPLEMENTATION

For the implementation of the GA, we used the Java-based framework presented in [34]. The GA's setup and parameters are shown in Table 2. GA is an evolutionary algorithm that maintains a population of solutions that keep evolving by applying genetic operators such as mutations and crossovers.

In the GA implementation, the population is initially populated by the specified number of feasible allocations. If this is not possible within a time limit of one minute, the program returns with an empty set of allocations indicating that the allocation problem is infeasible. An allocation is represented as an array of length n, where n is the number of components. The elements in the array have values in the range 0 to $m-1$, where m is the number of computational units. An element in the array at index i with value j means allocating the component $i+1$ to the computational unit $j+1$.

For mutation, we used the swap algorithm. If an individual (allocation) in the population becomes infeasible after mutation, we only add the original individual. For crossover, we used the 2-point crossover algorithm. Similar to what is done in mutation, if an offspring in the crossover is infeasible, we keep the original parent. For termination, we used the condition that the specified number of generations is reached. We used the cost function w in comparing the allocations.

B. ANY COLONY ALGORITHM IMPLEMENTATION

ACO algorithms emulate the behaviour of ants in nature to solve hard combinatorial optimization problems [35]. ACO algorithms were originally proposed by Dorigi *et al.* [36]. ACO algorithms have been applied successfully in many academic and real-world applications to solve computationally challenging problems [37].

For the implementation of the ACO algorithm, we used Isula which is a Java framework for ACO algorithms [38].

TABLE 3. AS Settings.

Number of Iterations	**100**
Number of Ants	**2000**
Evaporation Ration (ρ)	**0.4**
Pheromone Importance (α)	**1**
Heuristic Importance (β)	**1**

The ACO algorithm implemented is the Ant System (AS). The AS's setup and parameters are shown in Table 3. To store the pheromone values, we used an $n \times m$ two-dimensional array. In our implementation, ants are allowed to generate infeasible solutions, however when an ant generates an infeasible solution, the constructed solution is penalized by assigning the value *Integer.MAX_VALUE* as its cost (in the *getSolutionCost* method). In addition, if the partial solution constructed by an ant becomes invalid, then the corresponding heuristic value is penalized by assigning the value *Integer.MAX_VALUE* to it in the *getHeuristicValue* method.

V. VALIDATION

For the validation, we apply our framework to find optimal allocations on several component allocation problem instances. We compare the GA and ACO implementations in terms of the optimal cost found and the execution time.

First, consider the following instance of the component allocation problem. The parameters of the problem instance are borrowed from [12]. These are based on the deployment problem of a vision-based software system on an actual autonomous underwater vehicle. The system consists of $n = 11$ components and $m = 4$ computational units. The components are: **1-UI** User Interface, **2-CH** Communication Handler, **3-MP** Message Parser, **4-MD** Manual Drive, **5-MM** Mission Manager, **6-MC** Movement Control, **7-V** Vision, **8-AC** Actuator Control, **9-SI** Sensors Layer 1, **10-S2** Sensors Layer 2, and **11-SF** Stream Filtering components. The computational units are: **1-mCPU** Multicore CPU, **2-FPGA** FPGA I, **3-FPGA** FPGA II, and **4-GPU** GPU. There are $l = 3$ resources: CPU, memory, and energy resources. Figure 2 shows the component resource consumption represented as three matrices, one for each resource dimension: (a) for the CPU resource, (b) for the memory resource, and (c) for the energy resource. The computational unit resource capacity matrix is given by:

$$R = \begin{bmatrix} 100 & 256 & 50 \\ 150 & 640 & 25 \\ 150 & 640 & 25 \\ 100 & 256 & 15 \end{bmatrix}$$

and the trade-off vector is given by:

$$F = \begin{bmatrix} 0.1557 & 0.0856 & 0.7095 \end{bmatrix}$$

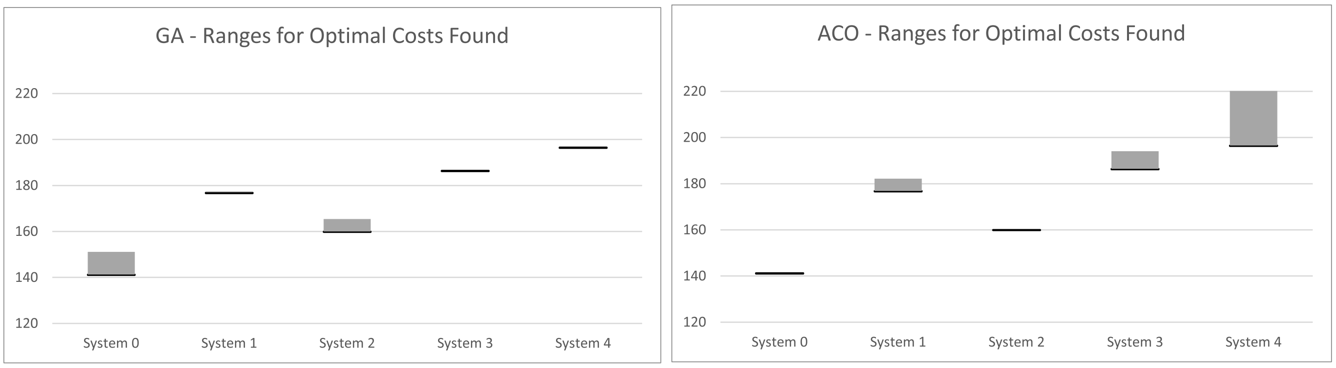
The following architectural constraints are considered:

- **Constraint I:** Component **7-V** should be allocated on **4-GPU**.
- **Constraint II:** Component **4-MD** should not be allocated on **1-mCPU**.

$$\begin{bmatrix} 10 & 90 & 90 & 55 \\ 50 & 20 & 20 & 72 \\ 30 & 20 & 20 & 72 \\ 10 & 40 & 40 & 72 \\ 20 & 40 & 40 & 72 \\ 20 & 50 & 50 & 55 \\ 90 & 20 & 20 & 15 \\ 20 & 10 & 10 & 70 \\ 20 & 10 & 10 & 70 \\ 20 & 15 & 15 & 70 \\ 90 & 10 & 10 & 33 \end{bmatrix} \qquad \begin{bmatrix} 48 & 256 & 256 & 128 \\ 128 & 256 & 256 & 148 \\ 64 & 256 & 256 & 148 \\ 48 & 168 & 168 & 148 \\ 64 & 168 & 168 & 148 \\ 64 & 168 & 168 & 64 \\ 168 & 128 & 128 & 64 \\ 148 & 96 & 96 & 148 \\ 48 & 32 & 32 & 148 \\ 48 & 32 & 32 & 148 \\ 168 & 64 & 64 & 96 \end{bmatrix} \qquad \begin{bmatrix} 2 & 18 & 18 & 11 \\ 10 & 4 & 4 & 14 \\ 6 & 4 & 4 & 14 \\ 2 & 8 & 8 & 14 \\ 4 & 8 & 8 & 14 \\ 4 & 10 & 10 & 11 \\ 18 & 4 & 4 & 3 \\ 4 & 2 & 2 & 14 \\ 4 & 2 & 2 & 14 \\ 4 & 3 & 3 & 14 \\ 18 & 2 & 2 & 7 \end{bmatrix}$$

(a) (b) (c)

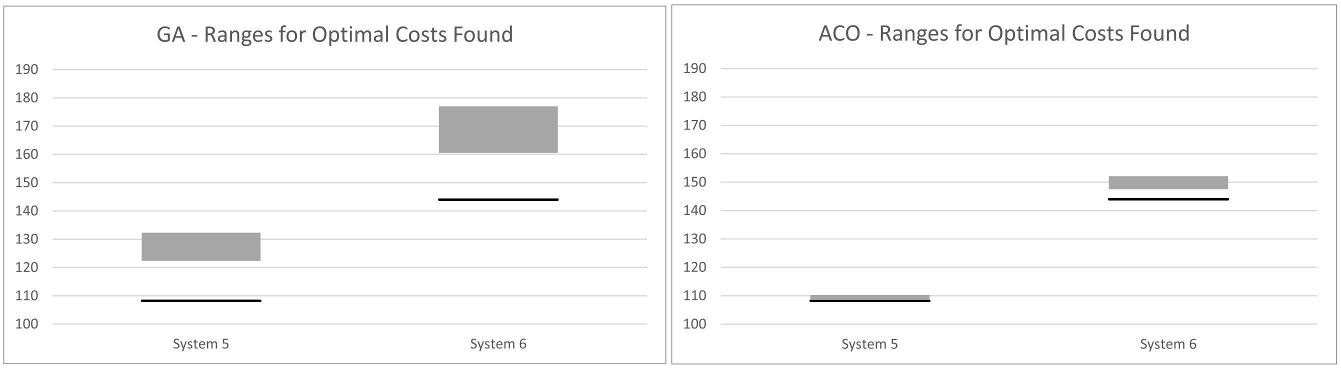
FIGURE 2. The component resource consumption matrix *T*.



(a) GA optimal cost results

(b) ACO optimal cost results

FIGURE 3. Optimal cost results for *System 0* through *System 4*.



(a) GA optimal cost results

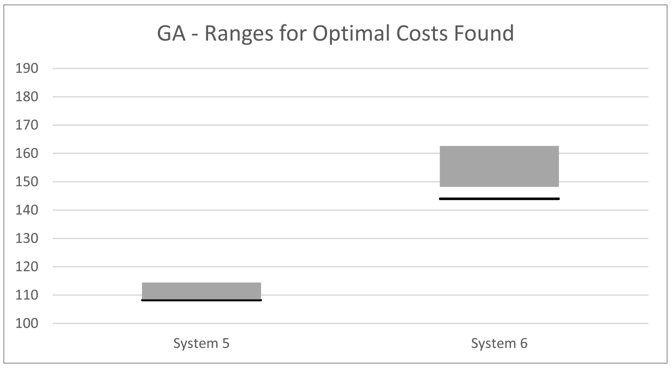
(b) ACO optimal cost results

FIGURE 4. Optimal cost results for *System 5* and *System 6*.

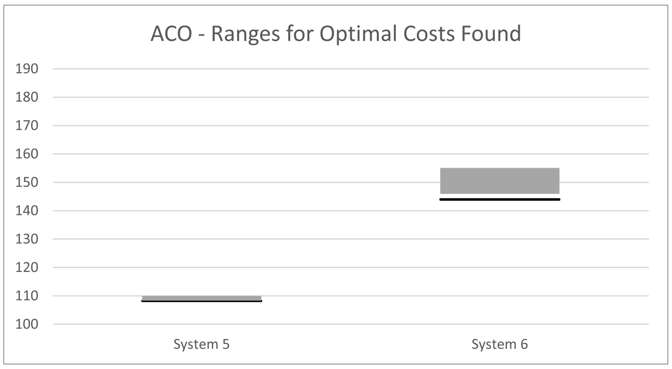
We will refer to the system consisting of the aforementioned components and computation units as *System 0*. In addition, we created several systems by swapping many elements with other randomly chosen elements in the same dimension in T. The same R and F matrices and the same constraints are used in these systems. In addition to *System 0*, we present the results for four systems denoted as *System 1* through *System 4*. Note that these systems have been used in the experiments conducted in earlier work on component allocation by one of the authors [4], [39], [40].

In the first set of experiments, we run the GA and ACO programs on each system for five times each. Each run returns the optimal cost as found by the corresponding program. In addition, we developed a program that implements an exhaustive search algorithm to find the optimal cost. The program can find optimal costs for component allocation problems with small search state spaces such as the aforementioned systems. However, for problems with large search state spaces such as the systems to be presented later, exhaustive search becomes infeasible.

Figure 3 shows the approximations to the optimal costs as found by the GA and ACO programs on the different runs. The same settings are used as those in Tables 2 and 3. The figure shows for each system the optimal cost obtained by exhaustive search as a black solid line. As stochastic solvers, since every run of the GA and ACO programs can result in

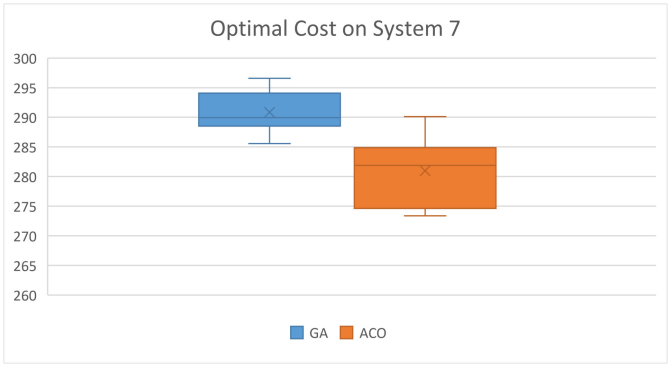


(a) GA optimal cost results

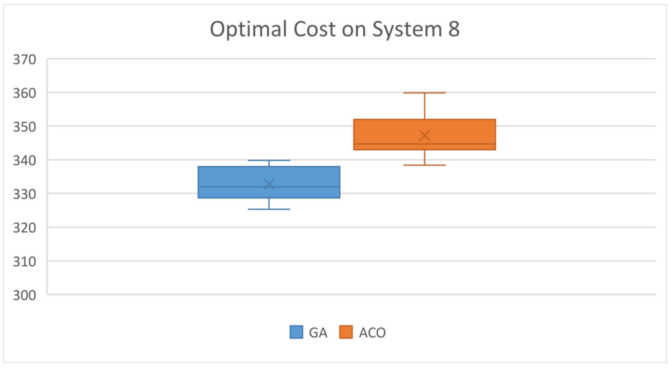


(b) ACO optimal cost results

FIGURE 5. Optimal cost results for *System 5* and *System 6*.



(a) Optimal cost results for *System 7*



(b) Optimal cost results for *System 8*

FIGURE 6. Optimal cost results for *System 7* and *System 8*.

a different value, the ranges of the obtained optimal costs are represented as gray rectangles delimited by the minimum and maximum values in the five runs for each system. The figure demonstrates that good approximations to the optimal costs are found by the GA and ACO programs. In addition, the figure demonstrates the need to run each solver several times in order to choose the minimum value as an approximation to the optimal cost. Also, considering the range widths, the performance of the GA program is worse than that of the ACO program on *System 0* and *System 2*. Conversely, its performance is better on *System 3* and *System 4*.

To study the performance of the GA and ACO programs on larger search state spaces, we created several new systems with larger values for n and m. These new systems were formed based on *System 0* through *System 4* by creating several copies of the computational units and creating a new component resource consumption matrix T based on the smaller base systems. For example, *System 5* is based on combining *System 0* and *System 1*. Four computational units are used in each of these systems, and the resource capacity matrix R is identical in both systems. For *System 5*, we used two copies of each computational unit with a total of $m = 8$. The number of components n is set to 11 similar to *System 0* and *System 1*. To create the component resource consumption matrix T for *System 5*, we assumed that the components' resource consumptions on the first four computational units

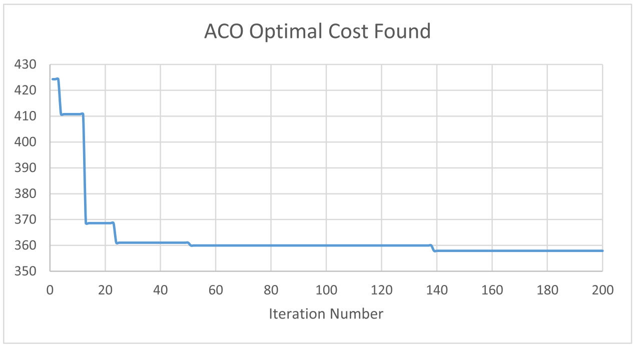


FIGURE 7. Time series depicting the ACO optimal cost found during the program' run.

are the same as those in *System 0* while for the last four computational units they are the same as those in *System 1*. With these new values for n and m, the search state space becomes too large with a total number of $m^n = 8^{11}$ allocations. Using exhaustive search was still feasible in this case, however its execution time was more than 19 minutes on the machine used in the experiment. More on the experiments to analyze the executions times of the ACO and GA programs is to be presented later in this section.

Figures 4 and 5 show the performance results of the GA and ACO programs on *System5* and *System 6*. *System6* was created in an identical manner as *System 5*, but rather

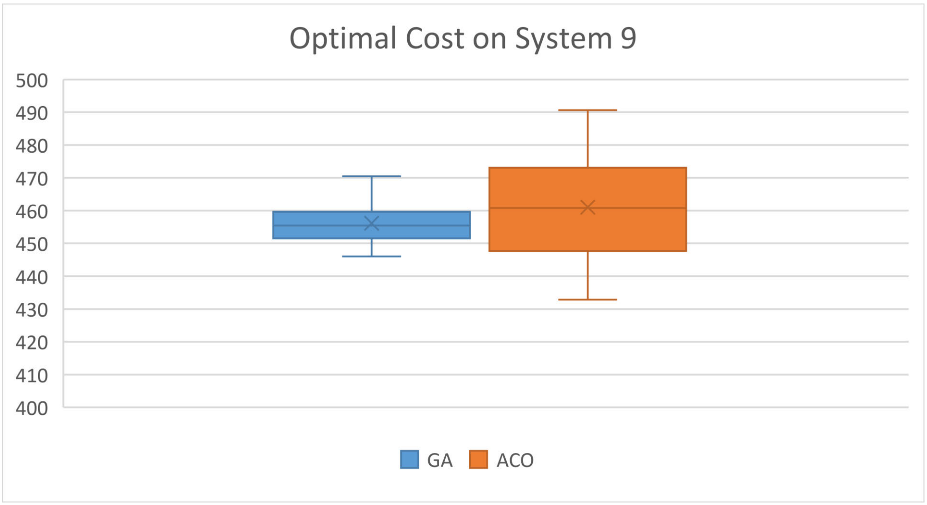


FIGURE 8. Optimal cost results for *System 9*.

by combining *System 3* and *System 4*. Figure 4 shows the results using the following updated settings: number of generations = 300 for the GA program and number of iterations = 100 for the ACO program. For Figure 5, the following updated settings are used: number of generations = 10000 for the GA program and number of iterations = 200 for the ACO program. The objective here is to re-examine the performance of the programs under different values for the settings. The figures demonstrate that the ACO program outperformed the GA program on these two systems with regards to how close the found approximations are to the optimal costs and how wide the ranges are.

Figure 6 compares the optimal costs found by the GA and ACO programs on two larger systems than the previous systems: *System 7* and *System 8*. In each system, there are $m = 16$ computational units and $n = 22$ components. *System 7* was formed by combining *System 0* through *System 3*, while *System 8* was formed by combining *System 1* through *System 4*. With these values for m and n, it becomes infeasible to use the exhaustive search algorithm to find the optimal cost. In such cases, it becomes necessary to use meta-heuristics such as GA and ACO algorithms.

Figure 6a contrasts the box plots for the optimal cost approximations found by the GA and ACO programs on *System 7*. In addition, Figure 6b contains the box plots for the optimal cost approximations found by the GA and ACO programs on *System 8*. Each box plot is based on a sample of 10 outputs returned by each program. Figure 6a demonstrates that the ACO program outperformed the GA program on *System 7*, however this was reversed in the case of *System 8* as demonstrated by Figure 6b. In both systems, we used the following updated settings: number of generations = 10000 for the GA program and number of iterations = 200 for the ACO program.

The execution times of the ACO program were observed to be several times larger than that of the GA program on *System 7* and *System 8*. For *System 7*, the 95% confidence interval for the execution times was 15.56 ± 0.95 (the unit is in seconds) in the case of the GA program and 342.15 ± 9.88 in the case of the ACO program. For *System 8*, the 95% confidence interval for the execution times was 15.00 ± 0.25 in the case of the GA program and 362.73 ± 30.39 in the case of the ACO program. However, we observe that the ACO program quickly converges to the best approximation for the optimal cost in the early iterations and usually it does not require a very large number of iterations. For example, Figure 7 is a time series depicting the ACO optimal cost found during the program' run on *System 8*. It shows that the ACO program generated a value (361.05) very close to the best optimal cost approximation (357.90) by the end of the 24th iteration. Note that the programs were were run on a desktop computer equipped with a 3.70GHz dual-core processor and 8GB RAM.

Figure 8 shows the box plots for the optimal cost approximations found by the GA and ACO programs on *System 9*. *System 9* is formed by combining *System 7* and *System 8*, with $m = 32$ and $n = 30$. Each box plot is based on a sample of 10 outputs returned by each program. As the figure demonstrates, the ACO program was able to obtain a better approximation for the optimal cost than the GA, however its range of the returned values is wider than that of the GA program. The best optimal cost approximation returned by the ACO program was 432.854, but for the GA program it was 446.0426. Also, the execution time of the ACO program (678.28 ± 73.76 seconds) is several times larger than the execution time of the GA program (76.91 ± 4.84 second). The same settings were used for *System 9* as in *System 7* and *System 8*.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented a framework for defining and solving software component allocation problem instances. The framework is based on the use of a meta-model defining the component allocation problem. The framework is implemented on top of Eclipse EMF, and it is designed with extensibility in mind. New meta-heuristics can be implemented and added to the framework. Furthermore, this paper has presented two meta-heuristics for solving the component allocation problem: GA and ACO. The experiments show that these meta-heuristics are able to find good approximations to optimal costs in large problem instances within a short execution time.

In future work, this research can be extended in two ways. First, the component allocation meta-model can be expanded to include communication cost. In addition, other types of architectural constraints can be specified on the placement of components. Second, new meta-heuristics can be implemented using the presented framework. These can be made available online so other researchers can experiment with them and can propose better meta-heuristics for solving the component allocation problem.

The framework presented in this paper only considers the single-objective version of the component allocation problem. The cost of an allocation is the weighted sum of the different resources consumed by the allocation. It is of interest to expand the framework to support a multi-objective problem formulation in which the aim becomes to find a set of Pareto-optimal allocations. As presented in the Related Work section, several meta-heuristics for solving the multi-objective component allocation problem exist. We believe that it is straightforward to extend our framework to support the multi-objective problem formulation.

REFERENCES

[1] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 709–724, Oct. 2007.

[2] I. Sommerville, *Software Engineering*, 9th ed. Reading, MA, USA: Addison-Wesley, 2011.

[3] C. A. Szyperski, D. Gruntz, and S. Murer, *Component Software—Beyond Object-Oriented Programming*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2002.

[4] I. Al-Azzoni, "An improved coloured Petri net model for software component allocation on heterogeneous embedded systems," *J. Comput. Inf. Technol.*, vol. 26, no. 2, pp. 85–97, Jun. 2018.

[5] I. Švogor and J. Carlson, "SCALL: Software component allocator for heterogeneous embedded systems," in *Proc. Eur. Conf. Softw. Archit. Workshops ECSAW*, 2015, p. 66.

[6] I. Švogor, I. Crnković, and N. Vrček, "An extensible framework for software configuration optimization on heterogeneous computing systems: Time and energy case study," *Inf. Softw. Technol.*, vol. 105, pp. 30–42, Jan. 2019.

[7] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An extensible framework for improving a distributed software system's deployment architecture," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 73–100, Jan. 2012.

[8] U. Pohlmann and M. Hüwe, "Model-driven allocation engineering: Specifying and solving constraints based on the example of automotive systems," *Automated Softw. Eng.*, vol. 26, no. 2, pp. 315–378, Jun. 2019.

[9] S. Wang, J. R. Merrick, and K. G. Shin, "Component allocation with multiple resource constraints for large embedded real-time software design," in *Proc. 10th IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, May 2004, pp. 219–226.

[10] J. Hartmanis, "Computers and intractability: A guide to the theory of NP-completeness (Michael R. Garey and David S. Johnson)," *Siam Rev.*, vol. 24, no. 1, p. 90, 1982.

[11] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2009.

[12] I. Švogor, I. Crnković, and N. Vrček, "An extended model for multi-criteria software component allocation on a heterogeneous embedded platform," *J. Comput. Inf. Technol.*, vol. 21, no. 4, pp. 211–222, 2013.

[13] R. W. Saaty, "The analytic hierarchy process—What it is and how it is used," *Math. Model.*, vol. 9, nos. 3–5, pp. 161–176, 1987.

[14] *Eclipse Modeling Framework (EMF)*. Accessed: Jun. 2020. [Online]. Available: https://www.eclipse.org/modeling/emf/

[15] *Graphical Modeling Framework (GMF)*. Accessed: Jun. 2020. [Online]. Available: https://www.eclipse.org/gmf-tooling/

[16] P. Feiler, D. Gluch, and J. Hudak, "The architecture analysis and design language (AADL): An introduction," Softw. Eng. Inst., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-2006-TN-011, 2006.

[17] J. Feljan, J. Carlson, and T. Seceleanu, "Towards a model-based approach for allocating tasks to multicore processors," in *Proc. 38th Euromicro Conf. Softw. Eng. Adv. Appl.*, Sep. 2012, pp. 117–124.

[18] H. Yamachi, H. Yamamoto, Y. Tsujimura, and Y. Kambayashi, "A solution method employing a multi-objective genetic algorithm to search for Pareto solutions of series-parallel system component allocation problem," in *Proc. IEEE Congr. Evol. Comput.*, Sep. 2007, pp. 3058–3064.

[19] A. Koziolek, H. Koziolek, and R. Reussner, "PerOpteryx: Automated application of tactics in multi-objective software architecture optimization," in *Proc. Joint ACM SIGSOFT Conf. QoSA ACM SIGSOFT Symp. ISARCS Qual. Softw. Archit. QoSA Architecting Crit. Syst. ISARCS QoSA-ISARCS*, 2011, pp. 33–42.

[20] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *J. Syst. Softw.*, vol. 82, no. 1, pp. 3–22, Jan. 2009.

[21] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Trans. Softw. Eng.*, vol. 35, no. 2, pp. 148–161, Mar. 2009.

[22] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II," *Proc. Int. Conf. Parallel Problem Solving Nature*, 2000, pp. 849–858.

[23] R. Li, R. Etemaadi, M. T. M. Emmerich, and M. R. V. Chaudron, "An evolutionary multiobjective optimization approach to component-based software architecture design," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jun. 2011, pp. 432–439.

[24] J.-F. Tilman, R. Sezestre, and A. Schyn, "Simulation of system architectures with AADL," in *Proc. Eur. Congr. Embedded Real Time Softw. Syst.*, Toulouse, France, 2008.

[25] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya, "ArcheOpterix: An extendable tool for architecture optimization of AADL models," in *Proc. ICSE Workshop Model-Based Methodologies Pervasive Embedded Softw.*, May 2009, pp. 61–71.

[26] *OSATE (Open Source AADL Tool Environment)*. Accessed: Jun. 2020. [Online]. Available: https://osate.org

[27] A. Aleti, L. Grunske, I. Meedeniya, and I. Moser, "Let the ants deploy your software—An ACO based deployment optimisation strategy," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2009, pp. 505–509.

[28] A. Ashraf, B. Byholm, and I. Porres, "A multi-objective ACS algorithm to optimize cost, performance, and reliability in the cloud," in *Proc. Int. Conf. Utility Cloud Comput.*, Dec. 2015, pp. 341–347.

[29] G. Campeanu, J. Carlson, and S. Sentilles, "Component allocation optimization for heterogeneous CPU-GPU embedded systems," in *Proc. 40th EUROMICRO Conf. Softw. Eng. Adv. Appl.*, Aug. 2014, pp. 229–236.

[30] G. Campeanu, J. Carlson, and S. Sentilles, "Allocation optimization for component-based embedded systems with GPUs," in *Proc. 44th Euromicro Conf. Softw. Eng. Adv. Appl. (SEAA)*, Aug. 2018, pp. 101–110.

[31] *CPLEX Optimizer*. Accessed: Jun. 2020. [Online]. Available: https://www.ibm.com/analytics/cplex-optimizer

[32] K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J. Lundback, and K.-L. Lundback, "The rubus component model for resource constrained real-time systems," in *Proc. Int. Symp. Ind. Embedded Syst.*, Jun. 2008, pp. 177–183.

[33] G. Campeanu and M. Saadatmand, "Run-time component allocation in CPU-GPU embedded systems," in *Proc. Symp. Appl. Comput. SAC*, 2017, pp. 1259–1265.

[34] L. Jacobson and B. Kanber, *Genetic Algorithms Java Basics*, 1st ed. Berkeley, CA, USA: Apress, 2015.

[35] C. Blum, "Ant colony optimization: Introduction and recent trends," *Phys. Life Rev.*, vol. 2, no. 4, pp. 353–373, Dec. 2005.

[36] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: Optimization by a colony of cooperating agents," *IEEE Trans. Syst. Man, Cybern. B, Cybern.*, vol. 26, no. 1, pp. 29–41, Feb. 1996.

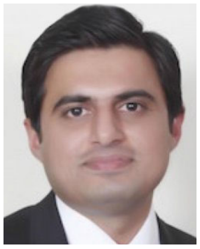
[37] M. Dorigo and T. Stützle, "Ant colony optimization: Overview and recent advances," in *Handbook of Metaheuristics*, 3rd ed., M. Gendreau and J.-Y. Potvin, Eds. Cham, Switzerland: Springer, 2019, pp. 311–351.

[38] C. Gavidia-Calderon and C. B. Castañon, "Isula: A java framework for ant colony algorithms," *SoftwareX*, vol. 11, Jan./Jun. 2020, Art. no. 100400.

[39] I. Al-Azzoni, "Software component allocation on heterogeneous embedded systems using coloured Petri nets," in *Proc. Conf. Adv. Trends Softw. Eng.*, 2015, pp. 23–28.

[40] L. Al-Dakheel and I. Al-Azzoni, "Model-to-model based approach for software component allocation in embedded systems," in *Proc. 5th Int. Conf. Model-Driven Eng. Softw. Develop.*, 2017, pp. 321–328.

ISSAM AL-AZZONI received the Ph.D. degree in software engineering from McMaster University, Ontario, Canada. He is currently with the College of Engineering, Al Ain University, Al Ain, UAE. His research interests include modeling, model transformation, and the application of formal methods in software engineering.



SAQIB IQBAL received the M.Sc. degree in software engineering from the Queen Mary University of London, U.K., in 2007, and the Ph.D. degree in software engineering from the University of Huddersfield, in 2013. He has taught in various postgraduate universities for over ten years and has worked in the industry as a Software Engineer for more than three years. He is actively involved in research in areas particularly related to requirements engineering, software design, model-transformation, and software testing.