

Parallel Constraint Handling in a Multiobjective Evolutionary Algorithm for the Automotive Deployment Problem

James Montgomery and Irene Moser
 Faculty of Information and Communication Technologies
 Swinburne University of Technology
 Melbourne, Australia
 Email: {jmontgomery, imoser}@swin.edu.au

Abstract—The component deployment problem is a complex multiobjective optimisation task faced by engineers in the automotive industry. Thus far, the best-known solutions to this problem have been achieved using the NSGA-II algorithm combined with a constraint handling method based on repairing solutions that have been rendered infeasible by the genetic operators. It can reasonably be assumed that an approach that repairs solutions immediately after a change has limited coverage of the infeasible space. Exchanging solutions with other algorithms may help enhance the search space coverage. However, we observe an improvement in performance through parallelisation only after increasing the complexity of the problem.

Index Terms—Automotive deployment; parallel optimisation; multiobjective problems; constraint handling.

I. INTRODUCTION

The automotive industry takes advantage of mass production by fitting vehicles of the same type or series with the same pre-designed infrastructure. In terms of on-board electronics, vehicles from the same series typically have the same arrangement of Electronic Control Units (ECUs) and data buses. ECUs are comprised of a processing unit (CPU) and a memory. The hardware infrastructure of a contemporary car usually consists of 50–80 ECUs, each of which is connected to one of the 3–5 data buses. Software components that control different aspects of functionality of a vehicle can be assigned to many different ECUs subject to memory requirements, processing power and similar restrictions. The software deployment problem [8], [10] seeks to optimise the deployment of these control logic components to ECUs. The problem is highly constrained by restrictions on which ECUs may host which components, the limited memory available on ECUs, as well as prescribed colocation constraints that either require or preclude pairs of components from residing on the same ECU.

The problem is multi-faceted in the sense that there are many different available objective functions, some of which are partly correlated. The infrastructure is usually fixed over a period of time; vehicles typically retain their hardware layout until a new series is launched. A similar problem to the software deployment problem is known as redundancy allocation [2], [6], where the goal is to introduce redundant

ECUs fulfilling identical functions to increase the reliability of the system.

In previous work [12] the software deployment problem was posed as a biobjective optimisation problem with the reliability of data communications between components as one and the communications overhead as a second objective. Due to the possible correlation between communications overhead and communications reliability, in this work the second objective is replaced by the average time taken by ECUs to execute each of their hosted components, which impacts on the responsiveness of the system. Additionally, as different constraint handling mechanisms can lead the solver to explore different regions of the solution space the NSGA is applied in parallel as part of an island model.

II. AN AUTOMOTIVE DEPLOYMENT PROBLEM

The software deployment problem described here represents one aspect of the optimisation task of allocating the software components that provide the control logic for contemporary vehicles (e.g., ABS and airbag control functions) to processing units distributed around the vehicle. The necessary software functionality is represented by a predefined set of software components, which have to be deployed to the ECUs. A solution to the software deployment problem is a mapping of all available software components C to all or a subset of ECUs U : $d_i = \{(c_1, u_{i_1}), (c_2, u_{i_2}), \dots, (c_n, u_{i_m})\}$, where i_1 to i_m are integers in $[1, m]$. The set of all possible solutions is $D = \{d \mid d : C \rightarrow U\}$. For ease of notation, the set of indices of components assigned to ECU u_j is denoted C_{u_j} . More in-depth description of the problem and its aspects is found in [1], [6], [7].

A. Hardware Infrastructure

ECUs are connected to different data buses and vary in memory capacity, processing power and failure propensity. Data buses are characterised by different data rates and degrees of reliability. The speed and reliability of communication between two ECUs therefore depends on the data buses to which they are connected. More formally, the hardware architecture is defined in the following terms:

- The set of available ECUs $U = \{u_1, u_2, \dots, u_m\}$, $m \in \mathbb{N}$;
- ECU capacity, $cp : U \rightarrow \mathbb{N}$;
- ECU processing speed, $ps : U \rightarrow \mathbb{N}$, being how many million machine instructions are processed per second;
- ECU failure rate, $fr : U \rightarrow \mathbb{R}$;
- data rate of the preferred bus, $dr : U \times U \rightarrow \mathbb{N}$;
- network delay, $nd : U \times CU \rightarrow \mathbb{R}$;
- reliability of the preferred bus, $rel : U \times U \rightarrow \mathbb{R}$.

B. Software

The optimisation task is to allocate a number of predefined software components to the ECUs in the existing hardware infrastructure—the arrangement and type of ECUs and buses connecting them is predetermined and not subject to further optimisation. Each software component fulfils a predefined function in the vehicle. Components are defined by the following:

- The set of components, $C = \{c_1, c_2, \dots, c_n\}$, $n \in \mathbb{N}$;
- component size in memory, $sz : C \rightarrow \mathbb{N}$;
- estimated execution length, expressed as a fraction of one million machine instructions, $mi \in \mathbb{R}$;
- location restriction, $lr : C \rightarrow \mathcal{P}(U)$;
- colocation restriction, $coloc : C \times C \rightarrow \{1, -1\}$;
- data size sent over a link, $ds : C \times C \times S \rightarrow \mathbb{R}$;
- frequency of communication between two components, $freq : C \times C \rightarrow \mathbb{R}$;
- the communication link between two components i and j , $l_{ij} = (c_i, c_j)$.

In reality, a component may spend different portions of its processing time supporting various services. However, in the present work this has been simplified to the single execution length, mi .

C. Objective Functions

As this is a multi-faceted problem, the available range of objectives is large. Previous work [1], [12] considered two non-functional quality attributes: Data Transmission Reliability (DTR) as defined by Malek [9] and Communication Overhead (CO) following Medvidovic and Malek [10]. Reliability of the data transmission is a crucial quality attribute in a real-time embedded system, where important decisions are taken based on the data transmitted through the communication links. The Data Transmission Reliability (DTR) formulation we use has first been defined by Malek [9].

$$f_{DTR}(d) = \sum_{i=1}^n \sum_{j=1}^n freq(c_i, c_j) \cdot rel(d(c_i), d(c_j)) \quad (1)$$

As there is a possible positive relationship between good data transmission reliability and low communication overhead (maximising the colocation of frequently communicating components should achieve both objectives) in this study communication overhead has been replaced by *scheduling time* (ST), which is the average time for an ECU to complete the round-robin processing of all its assigned components. ST is given by

$$f_{ST}(d) = \frac{1}{m} \cdot \sum_{j=1}^m \left(\frac{\sum_{i \in C_{u_j}} mi_i}{ps_j} \right). \quad (2)$$

Component responsiveness is an important aspect of an embedded system, and is affected by the number and nature of other components with which ECU processing time must be shared. This aspect of a deployment is measured by taking the average processing time required for ECUs to execute all of their allocated components in a round-robin fashion.

D. Constraints

Not all deployment candidates $d \in D$ represent feasible alternatives. The number of components to be placed on a single ECU is restricted by the memory size of the ECU and the memory requirements of the component. In analogy with the traditional bin packing problem, the memory constraint Ω_{mem} is defined as follows:

$$\Omega_{mem}(d) = \forall u \in U : \sum_{i \in C_{u_j}} sz(c_i) \leq cp(u_j) \quad (3)$$

The location constraint Ω_{loc} excludes certain components from residing on particular ECUs:

$$\Omega_{loc}(d) = \forall c \in C : (u \in lr(c) \Rightarrow d(c) \neq u) \quad (4)$$

The colocation constraint Ω_{coloc} excludes certain components from residing on particular ECUs while prescribing joint ECU allocations for some components:

$$\begin{aligned} \Omega_{coloc}(d) = & \forall c \in C : (c_i, c_j \in coloc_{-1} \Rightarrow d(c_i) \neq d(c_j)) \\ & \forall c \in C : (c_i, c_j \in coloc_1 \Rightarrow d(c_i) = d(c_j)) \end{aligned} \quad (5)$$

All of the constraints Ω are hard constraints in the sense that a solution that violates them cannot be considered for implementation. It is therefore meaningful to consider them separately from the objective functions.

III. ALGORITHMIC APPROACH

The Nondominated Sorting Genetic Algorithm (NSGA) in its improved form NSGA-II by Srinivas and Deb [4] is recognised as one of the most successful state-of-the-art multiobjective problem solvers. Previous work on the software deployment problem [12] successfully combined NSGA-II with three different constraint handling mechanisms. The same approach is used here to validate the previous results using a new objective and to form the basis of a parallelised implementation.

A. Solution Representation

In previous work, the usage of two alternative representations was studied. The representation illustrated in Fig. 1, a mapping from ECUs to a set of assigned components, was found to be more effective and is consequently the one used here.

c_1	c_5	c_1	c_7	c_2
c_4		c_6, c_8		
u_s	u_s	u_s	u_s	u_c

Fig. 1. Solution representation used by the genetic algorithm

B. Operators

Two mutation and one crossover operators have been implemented.

1) *Point Mutation*: A randomly selected component is reassigned to a randomly determined new ECU from the component's *lr* list of permissible ECUs subject to colocation constraints ($coloc_{-1}$). Random selections may have to be repeated to find an assignment that complies with location and colocation restrictions. Point mutation will only reassign a component that has no group members.

2) *Swap Mutation*: Unlike point mutation, the swap mutation operator randomly chooses a second component whose ECU assignment can be swapped with the first choice without violating the location and colocation constraints ($coloc_{-1}$). Swap mutations transfers all members of the same $coloc_{-1}$ group onto the new ECU.

3) *Crossover*: The non-standard crossover initially merges the lists of assigned components per ECU. In the second step, the components are divided between the same ECU on the first and the same ECU on the second child, subject to location and colocation restrictions. Algorithm 1 describes the operation.

C. Constraints

The constraints present in this problem are complex, and thus the generation of feasible solutions can be difficult [12]. Michalewicz and Schoenauer [11] categorised the existing approaches to constraint handling into penalty methods, methods which preserve feasibility, methods which distinguish between feasible and infeasible solutions and methods which repair solutions.

The assignment options provided for each component in the instance's location list, Equation (4), are a hard constraint which cannot be violated. The same applies to the colocation restrictions (Equation (5)). The genetic operators as implemented provide the possibility of transforming one feasible solution to another possible allocation without conceivable barriers between the solutions. These two constraints are therefore implemented as in Randall, Hendtlass and Lewis [13].

Memory constraint violations can easily be quantified in accordance with Equation (3) as the amount of memory requirements of the assigned components that exceeds the memory capacity of the ECU.

1) *Constrain-dominance for memory violation*: For the strategy of handling feasible and infeasible solutions separately we chose the implementation suggested by Deb [3], [4], and favoured by many authors. Constrain-dominated sorting compares feasible to feasible solutions according to fitness and infeasible solutions to infeasible solutions using the quantity

Algorithm 1: Crossover

Data: parent1_lst, parent2_lst
Result: offspring1_lst and offspring2_lst
Create list *ecu_lst* of length m and fill with empty lists;
for $i \in [1, m]$ **do**
 fill *ecu_lst*[i] with components from parent1_lst[i]
 and parent2_lst[i];
for $restarts < 2 * n$ **do**
 for $i \in [1, m]$ **do**
 for $j \in ecu_lst[i]$ **do**
 if $c_j \in coloc_1$ OR $c_j \in coloc_{-1}$ OR c_j is
 duplicate in *ecu_lst*[i] **then**
 if assignment to *offspring1_lst*[i] or
 offspring2_lst[i] can be made without
 violation **then**
 Assign c_j to *offspring1_lst*[i] or
 offspring2_lst[i];
 else
 restart;
 else
 With 50% probability add c_j from
 ecu_lst[i] to *offspring1_lst*[i], alternatively
 to *offspring2_lst*[i];
if still no new solution then
 throw exception;

of constraint violation. Feasible solutions always dominate infeasible solutions.

2) *Repairing memory violations*: According to the principle of feasibility preservation, an offspring that cannot be repaired after mutation or crossover will be discarded. The reassignment strategy used by the repair algorithm cannot always restore feasibility due to existing assignments which would have to be shifted first.

Algorithm 2: Repair Function

foreach ECU u_i that has allocations in excess of
memory capacity **do**
 while ECU u_i capacity exceeded **do**
 foreach Component c_j assigned to the ECU u_i
 do
 foreach ECU u_k in c_j 's list of ECU options
 do
 if u_k has enough memory and has no
 component that conflicts with c_j **then**
 assign $d(c_j, u_k)$;
 remove assignment $d(c_j, u_i)$;
 break;

3) *Penalising memory violation*: One of the more recent propositions to formulate a penalty function was conceived

by Woldeesenbet, Yen and Tessema [14]. It is based on the concept of the infeasible individual's distance to the feasible region. The higher the proportion of feasible solutions, the more relative significance is allocated to the fitness function, while the quantified constraint violation is considered to a lesser extent. If all solutions are infeasible, the fitness of the solutions is not considered at all. For details on the calculation of a penalty-adjusted fitness function, see [14].

IV. PARALLELISED SOLVER

Given that the three constraint handling approaches will likely explore different areas of the search space it may be beneficial to combine them. An island model allows for the transfer of solution traits between populations produced with different constraint handling approaches. A parallel version of the NSGA described above was implemented for execution on a commonly-available multiprocessor architecture. In this parallel algorithm, every k iterations the two separately evolving populations swap some of their solutions, the number and nature of which depends on which two constraint handling techniques are being used. Although any of nine combinations are possible, in this work only three combinations were examined, which fall into two categories: Repair with a constraint handling technique that allows infeasible solutions in the population, and two repair algorithms.

A. Combining Repair with Penalty and Constrain Dominance

The rationale for combining a repair-based technique with one that allows infeasible solutions (while applying selection pressure to move solutions towards feasibility) is that repair of solutions can work against exploration. In particular, if the solution space consists of 'islands' of feasibility then a search that requires that feasibility be maintained will find it difficult to move between those two islands [4]. The two parallel approaches in this category are Repair/Penalty and Repair/Constrain-dominance. Every k iterations, all feasible solutions from the either Penalty or Constrain-dominance population are swapped with the same number of feasible solutions from the first and, possibly, subsequent fronts of the Repair population.

B. Combining Repair with Repair

The other combination of algorithms consists of two Repair-based techniques. As both populations consist entirely of feasible solutions the swap mechanism employed with Penalty or Constrain-dominance cannot be used. Instead, every k iterations both populations are merged and then individuals randomly reallocated to one of the two populations. This mixing of individuals can thus serve to maintain diversity in the two populations and encourage exploration of different areas of the search space.

V. VALIDATION

A. Experimental set

Two problem instances were generated for the experimentation. The actual specifications dictate that the proportion

of ECUs to components is within predefined boundaries; the number of components should be slightly less than twice the number of components, leaving little room for variation within realistic settings. Contemporary hardware infrastructure has no more than 100 ECUs; problems with less ECUs are of significantly less combinatorial complexity and uninteresting from the point of view of optimisation. The generated problems are of realistic size with 140 components each and 85 and 90 ECUs respectively. The small difference in ECU numbers is justifiable and aims at augmenting the number of placement options for the algorithm, so that algorithm behaviour on different sizes of feasible space can be observed.

The problem instances allow an average of 16 (85 ECUs/140 components) or 17 (90 ECUs/140 components) placement choices for each component. The memory restrictions dictate that an average of 5 components can be located on the same ECU at any one time.

B. Algorithmic settings and parameters

For the experimental validation we chose two sets of experiments. The first consists of trials with two combinations of approaches, one of them parallelising repair-based constraint handling ('Repair') with Constrain Dominance ('Constrain dominance') in one instance and Repair with penalty-based constraint handling ('Penalty') in the second instance. This set of trials runs NSGA with each of the constraint-handling methods individually as a control group of three trials.

The parameters for the NSGA were chosen in accordance with findings in previous work [12]. They are a crossover rate of 1.0, a mutation rate of 0.5 with equal probability of using point or swap mutation and a population size of 50 individuals. The initial set of trials allows 200 iterations to all algorithms. Where algorithms were run in parallel, 200 iterations were used for each of the algorithms, which results in all runs taking equal amounts of time given all contemporary hardware has several CPU cores and any single trial takes less than 15 minutes. The solutions are swapped between populations every 50 iterations (i.e., after iterations 50, 100 and 150). All trials were repeated 30 times.

Finally, a second set of trials compares the performance of two parallelised Repair approaches with a 'conventional' repair-based NSGA. To establish the effects of additional iterations, these trials were run with 300 iterations.

C. Experimental Results

The individual runs of NSGA with Constrain Dominance, Penalty and Repair respectively as a control group corroborate the results discussed by Moser and Mostaghim [12]. The Repair approach is found to be superior to both Penalty and Constrain Dominance by a large margin. The latter produce infeasible solutions during crossover and mutation, but these solutions are usually discarded in the subsequent selection process. According to the Constrain Dominance rule, an infeasible solution never dominates a feasible one, hence the infeasible solutions will be discarded as soon as feasible ones are available. The largest part of the search space is infeasible, therefore

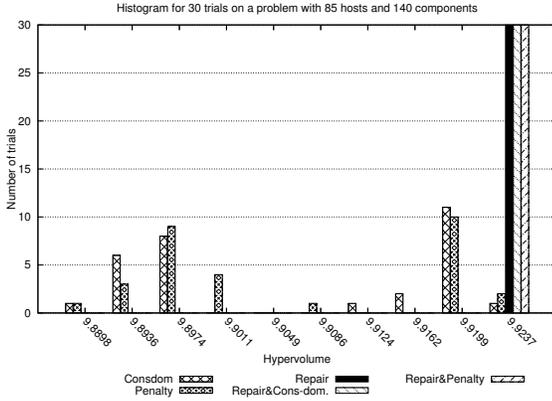


Fig. 2. Hypervolumes produced by the algorithms over 30 trials.

the optima are presumably located on islands. Discarding infeasible solutions quickly prohibits a traversal of these infeasible areas.

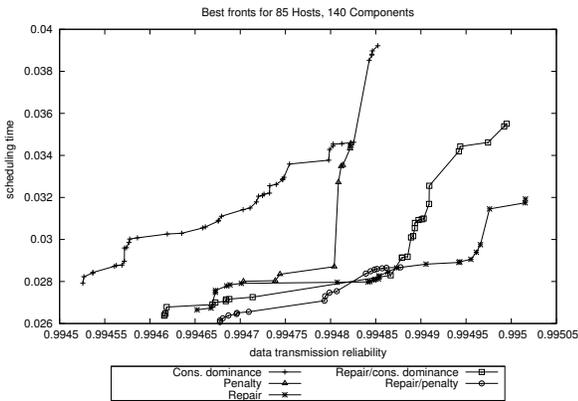


Fig. 3. Best approximation sets obtained in 30 runs of each algorithm using the instance with 85 ECUs and 140 components.

The hypervolume indicator is a commonly used measure for quality of approximation sets. It does not discern adequately between sets with mediocre but numerous and few but high-quality approximation sets [5]. Therefore, no numerical hypervolume measures are compared. In the absence of a more reliable metric, we use the hypervolume as an indicative measure to obtain some indication of the fluctuations in result quality. The approximation sets obtained from 30 trials are of very diverse quality in the case of the Constrain Dominance and Penalty approaches Fig. 2. The hypervolumes produced in 30 runs by the Repair approach as well as the parallel approaches (Repair/Cons. dominance and Repair/Penalty), also shown in Fig. 2 show far greater consistency.

In an attempt to confirm the quality indications of the hypervolume indicator, we plot the fronts it identifies as the best (Fig. 3) and worst (Fig. 4). The image showing the best fronts still shows reasonably good results for Constrain Dominance and Penalty, whereas the worst resulting fronts in

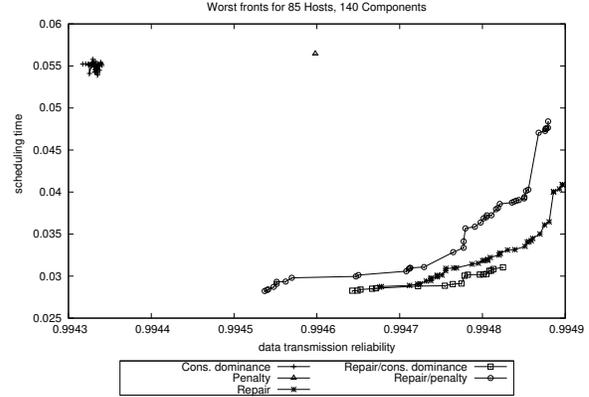


Fig. 4. Worst approximation sets obtained in 30 runs of each algorithm using the instance with 85 ECUs and 140 components.

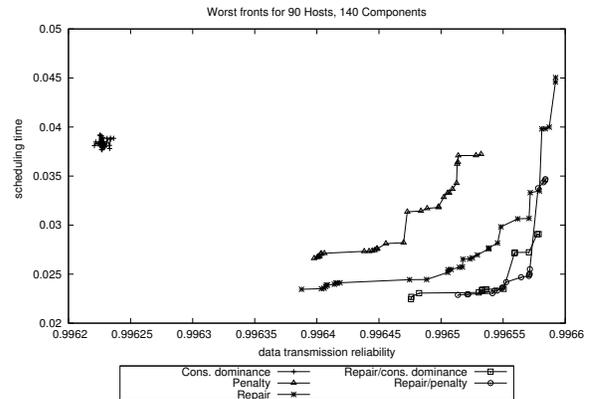


Fig. 5. Worst approximation sets obtained in 30 runs of each algorithm using the instance with 90 ECUs and 140 components.

some of the trials reduce to a single feasible solution in the case of the Penalty approach and to a tightly clustered set of ‘bad’ solutions for Constrain Dominance.

Even the worst solutions produced by the Repair and parallel approaches are relatively close to the best fronts, which indicates more reliable performance. However, it is difficult to determine superiority between the three approaches. The assumption that Constrain Dominance and Penalty would explore infeasible space and therefore provide more diverse solution sets to the Repair algorithm has not been confirmed. The algorithm seems capable of covering sufficient areas of the search space by immediately repairing a solution made infeasible by an operator.

Using a larger instance with 90 ECUs did change the behaviour of the algorithms slightly. Fig. 5 suggests that the parallelised algorithms now perform better than the Repair approach on its own. However, the hypervolumes (Fig. 6) obtained from these runs rather conclude in favour of the Repair algorithm. The slightly larger search space, however, reduces the difference in performance between the Constrain Dominance and Penalty approaches and the superior Repair

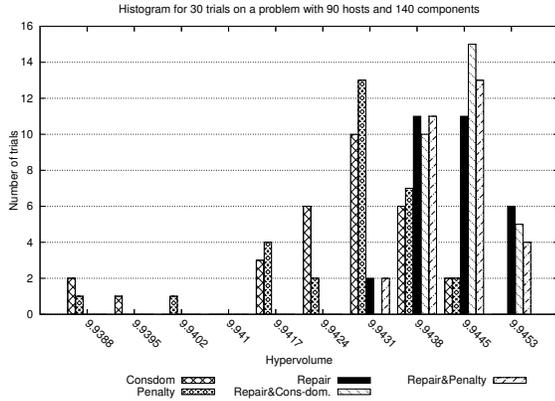


Fig. 6. Hypervolumes produced by the algorithms over 30 trials.

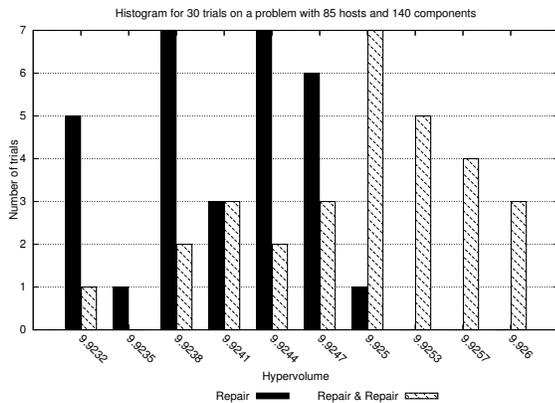


Fig. 7. Hypervolumes produced by the algorithms over 30 trials.

and parallelised methods. This seems to indicate that with growing number of possible assignments, ‘good’ areas of the search space become more accessible.

Given the performance of the Repair algorithm, it was judged meaningful to run two Repair-based algorithms in parallel such that they randomly swap solutions after every 30th iteration. The results of these trials and a comparison with the Repair approach used as before are shown in Fig. 7 for the more constrained problem with 85 ECUs. Both approaches use 300 iterations per algorithm. The comparison concludes clearly in favour of the parallelised approach. When fewer options are available, the Repair approach seems to focus the search on a subset of the possible solutions. The results suggest that there is a ‘constraint threshold’ above which it becomes useful to parallelise the Repair approach.

VI. CONCLUSIONS

The research described in this paper corroborates the outcomes of earlier work which indicated that repairing solutions immediately after they have been created by the genetic operators produces superior results to the approaches Constrain dominance proposed by Deb [3], [4] and a novel penalty function introduced by Woldesenbet et al. [14]. Both seemed

appropriate constraint handling methods for the automotive deployment problem. There was reason to assume that although the Penalty and Constrain dominance methods do not achieve the best approximation sets, their findings might be beneficial as inputs to the Repair-based NSGA as a means of diversifying the search space. However, on problem instances of realistic complexity, the optimising Repair-based NSGA does not yet seem to benefit from this effect. There are, however, some indications that parallelising the Repair-based NSGA with an identical implementation would result in better approximation sets as the problem complexity grows.

REFERENCES

- [1] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya, “ArcheOpterix: An extendable tool for architecture optimization of AADL models,” in *Model-based Methodologies for Pervasive and Embedded Software (MOMPES)*, 2009, pp. 61–71, 2009.
- [2] D. W. Coit and A. E. Smith, “Reliability optimization of series-parallel systems using a genetic algorithm,” *IEEE Transactions on Reliability*, vol. 45, no. 2, pp. 254–260, 1996.
- [3] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, “A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II,” Indian Institute of Technology, Kanpur, India, KanGAL report 200001, 2000.
- [4] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast elitist multi-objective genetic algorithm: Nsga-II,” *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2002.
- [5] E. Ducheyne, B. De Baets, and R. De Wulf, “Fitness inheritance in multiple objective evolutionary algorithms: A test bench and real-world evaluation,” *Applied Soft Computing*, vol. 8, no. 1, pp. 337 – 349, 2008.
- [6] L. Grunske, “Identifying “good” architectural design alternatives with multi-objective optimization strategies,” in *International Conference on Software Engineering ICSE*, 2006, pp. 849–852, 2006.
- [7] —, “Early quality prediction of component-based systems - A generic framework,” *Journal of Systems and Software*, vol. 80, no. 5, pp. 678–686, 2007.
- [8] S. Malek, “A user-centric approach for improving a distributed software system’s deployment architecture,” Ph.D. dissertation, Faculty of The Graduate School University Of Southern California, 2007.
- [9] —, “A user-centric approach for improving a distributed software systems deployment architecture,” PhD in Computer Science, Faculty of The Graduate School University of Southern California, 2007.
- [10] N. Medvidovic and S. Malek, “Software deployment architecture and quality-of-service in pervasive environments,” in *Workshop on the Engineering of Software Services for Pervasive Environments, ESSPE*. ACM, 2007, pp. 47–51, 2007.
- [11] Z. Michalewicz and M. Schoenauer, “Evolutionary algorithms for constrained parameter optimization problems,” *Evol. Comput.*, vol. 4, no. 1, pp. 1–32, 1996.
- [12] I. Moser and S. Mostaghim, “The automotive deployment problem: A practical application for constrained multiobjective evolutionary optimization,” in *2010 IEEE Congress on Evolutionary Computation (CEC 2010)*, 2010, pp. 4272–4279, 2010.
- [13] M. Randall, T. Hendtlass, and A. Lewis, “Extremal optimisation for assignment type problems,” in *Biologically-Inspired Optimisation Methods*, ser. Studies in Computational Intelligence, A. Lewis, S. Mostaghim, and M. Randall, Eds. Springer Berlin / Heidelberg, 2009, vol. 210, pp. 139–164, 2009.
- [14] Y. G. Woldesenbet, G. G. Yen, and B. G. Tessema, “Constraint handling in multiobjective evolutionary optimization,” *Trans. Evol. Comp.*, vol. 13, no. 3, pp. 514–525, 2009.