Fitness Landscape Characterisation for Constrained Software Architecture Optimisation Problems

Aldeida Aleti
Faculty of Information Technology
Monash University, Australia
Email: aldeida.aleti@monash.edu

Irene Moser
Faculty of Science, Engineering and Technology
Swinburne University of Technology
Email: imoser@swin.edu.au

***Abstract*—The automation of software architecture design is an important goal in software engineering. A plethora of automated design exploration techniques have been devised in the last decades to handle the complexity of making design decision in large scale, complex software systems. The common aim of these methods is the optimisation of quality attributes, such as reliability and safety. The majority of approaches use heuristic methods, such as local search or genetic algorithms, which use gradients in the fitness space to guide the search to the local optimum. When problems are constrained, search gradients are disrupted by infeasible regions, which may have a great impact on the difficulty of solving optimisation problems.**

Discovering the conditions under which a search heuristic will succeed or fail is critical for understanding the strengths and weaknesses of different software architecture optimisation methods. This paper investigates how to adequately characterize the features of constrained problem instances that have impact on difficulty in terms of algorithmic performance, and how such features can be defined and measured for the component deployment optimisation problem. We employ fitness landscape characterisation metrics that measure uniformity of the gradients in the search space, and investigate how two different constraints shape the search space, and as a result affect the performance of software architecture optimisation approaches.

Keywords*—*Software architecture optimisation; fitness landscape characterisation; reliability; constraints

I. Introduction

Software systems are present in many areas of our lives, such as medical, automotive, railway and telecommunication systems [30]. They usually perform complex functions that require high levels of reliability and safety. The design and development of today's software systems is a difficult task, especially when a great number of design options have to be considered. Different designs can be explored at the architectural level, where the embedded system is expressed formally using an architectural description language [13]. Software architecture has been defined as 'the fundamental concepts or properties of a system in its environment embodied in elements, relationships, and in the principles of its design and evolution' [18]. This view allows for the formulation of software architecture design as a search problem, defined as finding optimal design decisions for the given quality requirements.

Component deployment is a crucial design aspect, which deals with the allocation of software components to hardware hosts, and the assignment of interactions between components to the communication links. Deployment decisions have significant implications on the quality of the final system [15]. For instance, deploying frequently interacting components to different hosts entails a more frequent use of the network links, which affects the reliability of the system, defined as the continuity of correct service [7].

For many decades, sophisticated architecture optimisation techniques have been developed to deal with the complexity of software systems, the enormous design space and the effect of design decisions on quality attributes [3]. These efforts include methods like linear programming [31], [10], genetic algorithms [24], [1], [22], and local search [17]. The majority of these approaches consider experimental studies to determine the success of the optimisation strategy based on a set of selected problem instances. The no-free-lunch theorems tell us that 'for any algorithm, any elevated performance over one class of problems is exactly paid for in performance over another class' [35]. An investigation of the performance of optimisation algorithms on different problem instances, and how it generalises across different dimensions of the instance space shows that algorithms which perform best overall on a broadly defined collection of instances, are rarely the best on well-defined and more specific subsets of the instance space [11]. Hence, any elevated performance over one class of problems is exactly paid for in performance over another class. Studies that only focus on demonstrating the superiority of an architecture optimisation method over a set of other approaches are not sufficient for generalising the results to untested problem instances. Instead, more insightful conclusions can be drawn from the description of conditions under which an approach is expected to fail or succeed, which highlights the need in exploring the relationship between key characteristics of optimisation problem instances and algorithm behaviour [32].

The primary challenge in quantifying the relationship between algorithm performance and problem instance characteristics effectively is the explanatory power of the information considered. One successful approach to characterise the degree of difficulty a problem poses to a search algorithm has been to consider the search space and its properties, known as *fitness landscape analysis* [16]. Fitness landscape analysis aims at quantifying the relationship between the structure of the search space and the behaviour of search algorithms [33]. The analysis of this relationship is usually based on identifying challenging features of the fitness landscape, such as deception, multimodality and isolation [32].

Using the concept of fitness landscape, it is possible to study the structure of a search space, and analyse features that make problems hard to solve. This allows the study of the dynamics of the evolution of solutions, the comparative effectiveness of search methods, and the ability of the algorithms to find good solutions to a given problem. For instance, a fitness landscape with many local optima and an isolated global optimum can be deceptive and hard to search, since there is a high chance that the search algorithm gets stuck in a local optimum. Various fitness landscape characterisation metrics have been introduced to analyse the difficulty of optimisation problems. Predictive diagnostic optimisation [16] was devised to collect information about the difficulty of the search space while the search progresses. It has been applied to unconstrained problems and detects rugged search landscapes which make it difficult for the solver to follow gradients to optima.

In this paper, we investigate the features of the component deployment problem, with a specific focus on reliability optimisation and two constraints (memory and communication), and aim at finding out what makes this problem challenging to optimise. To this end, we identify and construct suitable hardness-revealing features for constrained problems. The problems we investigate are both complex and constrained. Problem instances of different degree of interactions among software components were generated (e.g. in some problem instances only 10% of components interact, whereas in the most constrained instance all components interact); the greater the interactions, the more constrained a problem would be. The goal is to detect problem difficulty that arises from the constraints the problem imposes, the fitness function and the search space. One of the most important questions that we seek to answer is whether the degree of interaction between components affect the performance of different optimisation techniques. The results from the experimental evaluation provide insights to inform algorithm selection for software architecture optimisation.

II. Related Work

Architecture specifications and models are used to structure complex software systems and provide a blueprint that is the foundation for later software engineering activities, such as software design [18], and support software engineers in coping with designing and developing complex software systems [3]. Thus, software architecture design is considered as one of the most important phases in the software development cycle [8], especially because design decisions affect the quality attributes of the final system [15]. Component deployment is one of the decisions that has to be made at an architectural level.

The component deployment problem has been formulated as a single-objective [14], [12], [15] or multiobjective optimisation problem [5], [4], [28], [29]. Some methods search for deployment architectures that satisfy constraints or user requirements [9], [23], other methods for optimal deployment architectures or at least candidates being close to it [15], often in combination with given constraints [15].

Aleti et al. [4] combined the optimisation of data transmission reliability with communication overhead, and compared the performance of a population based Ant Colony Optimisation with a multiobjective genetic algorithm. The approach considered three constraints: memory capacity, location and colocation constraints. Memory constraint was modelled as the restriction in the allocation of software components in the same hardware host if their total memory exceeds the memory of the host. In this paper we use a similar constraint. The location constraint restricts the allocation of certain software component to hardware hosts which are equipped with appropriate sensors. The colocation constraint does not allow the allocation of redundant software components to the same hardware host. Many other approaches have investigated localisation constraint [29], [12], [15] and memory constraint [14]. Other possible constraints are cost [28], [12] and timing [14], [15].

A nondominated-sorting genetic algorithm II (NSGA-II) was applied to the component deployment problem to investigate the performance of various constraint handling approaches [27], including penalty function, discarding infeasible solutions and using a repair function. The constraints considered are memory and location. The experimental evaluation showed that repairing infeasible solutions was by far the most successful method.

In later work, P-ACO and NSGA-II were applied to a triobjective variation of the component deployment problem under memory, location and colocation constraints [26]. The objectives considered were data transmission rate, communication overhead, and scheduling time, defined as a measure of system responsiveness. Unlike in the biobjective case [4], P-ACO outperformed NSGA-II when combined with a local search, which changed the allocation of a single component if this shift improved the hypervolume contribution of the solution. A problem specific heuristic which uses a Bayesian method to learns successful component-host allocations, and increases the probability of using them in the subsequent iterations of the algorithm [5]. Compared to NSGA-II and P-ACO, the Bayesian heuristic was found to produce approximation sets with higher hypervolume values.

The wide range of applications of search methods to the different problems in embedded system design has proven that no algorithm has optimal performance in all problems or problem instances. The successful application of a search method to a particular problem depends on the search space of that problem. The structure of the search space is shaped by the fitness function, constraints and the interaction between solution components. Ideally, we would select a search algorithm which explores the search space effectively, which requires means of measuring the search space. In this work, we make a first step in the direction of characterising the search space of problems encountered in embedded system design optimisation by investigating the component deployment problem.

III. Component Deployment

Formally, the set of software components is denoted as $\mathcal{C} = \{c_0, c_1, ..., c_n\}$, where $n \in \mathbb{N}$. The software components are considered as not modifiable and with unknown internal structure, but with a description of externally visible parameters. Each software component is annotated with the following properties: i) sz_i: the size of the memory in kilobytes (KB) that component c_i requires when deployed to a hardware host, ii) wl_i: the computational load in million instructions (MI) of component c_i, iii) q_i: the probability that the execution of the system starts at component c_i.

The software system starts executing in one software component (with a certain probability q_i), and during its life-time uses many other components through communication links. The transfer of execution from one component to another is known as interaction. Interacting components that are deployed to different hosts must use the communication links (buses) for successful execution. Interactions have a certain probability of happening, and are associated with a transition probability. This view follows the Kubat model [19], who expresses the software architecture as a Discrete-Time Markov Chain (DTMC), where vertices represent the execution of software components, and arcs enumerate the probability of transferring the execution flow to the next component. Software interactions are specified for each link from component c_i to c_j, which are annotated with the probability that component c_j is executed after component c_i, denoted as p_{ij}.

The software architecture is deployed to the hardware architecture, which is composed of a distributed set of hardware hosts, denoted as $\mathcal{H} = \{h_0, h_1, ..., h_m\}$, where $m \in \mathbb{N}$. Hardware hosts have different capacities of memory, processing power, and failure rates, which can be summarised as follows: i) fr_i: failure rate characterises the probability that hardware host h_i fails [6], ii) cp_i: memory capacity in kilobytes (KB) of the hardware host c_i, iii) ps_i: the instruction-processing capacity of hardware host h_i, expressed in MIPS (million instructions per second). This is used to calculate the execution time, which is a function of the processing speed of the hardware unit and the computation workload of the service.

The communication between components deployed to different hardware hosts is possible through the network links that connect them. The properties of network links are defined as follows: i) dr_{ij}: the data transmission rate in kilobytes per second (KBPS) of the network link that connects hosts h_i and h_j. Data rate is part of the calculation of the data transmission, ii) fr_{ij}: failure rate characterises the data communication failure of network link that connects hosts h_i and h_j.

The way the components are deployed into the hardware hosts affects many aspects of the final system, such reliability [25], [4], [28], [15]. Formally, the component deployment problem is defined as $D = \{d \mid d : \mathcal{C} \rightarrow \mathcal{H}\}$, where D is the set of all functions assigning components to hardware resources.

One may argue that real distributed embedded systems do not have the luxury of permitting an optimisation algorithm to decide where software components go. For instance, latency requirements often force certain components to be near the hardware they service, and software components cannot just be moved to a different CPU, since there are often architecture dependencies, operating system dependencies, etc. woven in the structure of the software. In addition, components vary widely in terms of their resource requirements, meaning that there will often be little freedom in where to run the larger components. All these aspects can be considered during the optimisation process, either by including constraints regarding the allocation of software components on specific hardware, or by introducing heuristics which guide the optimisation algorithm to make feasible choices.

A. Reliability Estimation

The reliability of a deployment architecture is calculated using a Markov model [34], which is a technique used for analysing complex probabilistic systems by abstracting the system behaviour to a set of mutually exclusive states and transitions between the states [21]. A Markov model is uniquely determined by a set of equations that describe the probabilistic transitions among the states and initialisation probability distributions of the starting states. The state transitions in Markov models are independent of the history, i.e. transition from state *i* to state *j* only depends on the state *i*, independently from how the state i was reached. This means that the complete history of reaching a system state is summarised in every state.

More specifically, we use a Discrete Time Markov Chain (DTMC) represented by finite state machines annotated with probabilities attached to transitions. A DTMC is used to represent possible states of a software execution and the probability to move from one state to another, starting from an *initial state* and ending either with a *successful* completion of execution or *failure*.

Formally, a DTMC can be expressed as a tuple (S, s_0, P) where, S is a finite set of states, $s_0 \in S$ is the *initial state*, and $P : S \times S \rightarrow [0, 1]$ is the *transition probability matrix*, where $P(s, s')$ denotes the probability of making a transition from state s to the state s'. In a DTMC, $\sum_{s' \in S} P(s, s') = 1$ for all states $s \in S$, which implies that even terminating states should have an outgoing transition to themselves with a probability of 1. When a system is modelled as a DTMC, the execution is represented by a path through the DTMC.

The DTMCs is constructed from the behavioural specification of the system such that a node represents the execution of one software component and arcs denote the transfer of execution from one component to another. *Super-initial* nodes are added to represent the start of the execution, and arcs are added from those nodes annotated with relevant execution initialisation probabilities (q). The model assumes that the components fail independently and the reliability of the component i is characterised by the probability r_i that the component performs its function correctly, computed as

$$r_i = e^{-\text{fr}_{d(c_i)} \cdot \frac{\text{wl}_{c_i}}{\text{ps}_{d(c_i)}}} \tag{1}$$

where $d(c_i)$ denotes the hardware host where component c_i is deployed. The reliability of a communication element is characterised by the failure rates of the network links and the time taken for the communication, defined as a function of the bus data rates dr and data sizes ds required for components c_i and c_j to communicate, defined as

$$r_{ij} = e^{-\text{fr}(d(c_i), d(c_j)) \cdot \frac{\text{ds}(c_i, c_j)}{\text{dr}(d(c_i), d(c_j))}}. \tag{2}$$

Next, the *expected number of visits* of each component $v : C \rightarrow \mathbb{R}_{\geq 0}$ is calculated as

$$v_i = q_i + \sum_{j \in \mathcal{I}} v_j \cdot p_{ji}. \tag{3}$$

The expected number of visits of a DTMC node quantifies the expectation of a component being used during a single system execution. The transfer probabilities $p_{i,j}$ can be written in a matrix form $P_{n \times n}$. Similarly, the execution initiation probabilities q_i^0 can be expressed with matrix $Q_{n \times 1}$. The

matrix of expected number of visits for all components $V_{n\times 1}$ can be calculated as:

$$V = Q + P^T \cdot V \tag{4}$$

Using matrix operations, eq. 4 can be transformed to:

$$V = (I - P^T)^{-1} \times Q \tag{5}$$

where I denotes the identity matrix. For absorbing DTMCs, the inverse matrix $(I-P^T)^{-1}$ always exists [34], which makes it possible to compute matrix V.

Next, the expected number of visits of network links $v_l : C \times C \rightarrow \mathbb{R}_{\geq 0}$ is calculated, where $v_l(c_i, c_j)$ denotes the expected number of occurrences of the transition (c_i, c_j). To compute this value, each probabilistic transition $c_i \xrightarrow{p_{ij}} c_j$ in the model is considered as a tuple of transitions $c_i \xrightarrow{p_{ij}} l_{ij} \xrightarrow{1} c_j$, the first adopting the original probability and the second having probability $=1$. Since the execution is never initiated in a link l_{ij} and the only predecessor of link l_{ij} is component c_i, the expected number of visits of a communication link is equal to

$$v_{ij} = v_i \cdot p_{ij} \tag{6}$$

Using expected number of visits and reliabilities of execution and communication elements, the reliability of a deployment architecture $d \in D$ is calculated as

$$R \approx \prod_{i\in\mathcal{I}} R_i^{v_i} \cdot \prod_{i,j\in\mathcal{I}} R_{ij}^{v_{ij}} \tag{7}$$

where $\mathcal{I}$ denotes the index set of all nodes in the DTMC.

B. Constraints

The component deployment problem is inherently constrained. Not all components can be deployed into the same hardware unit due to memory restrictions. At the same time, if interacting components are deployed into different hardware units, one has to make sure that there is a communication channel between them. In this paper we consider two constraints, memory and communication. First, we shall define a binary variable $x_{ij} \forall i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}$, such that

$$x_{ij} = \begin{cases} 1 & \text{if component } i \text{ is deployed into host } j \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

1) Hardware memory capacity: This constraint deals with the memory requirements of software components and makes sure that there is available memory in the hardware units. Processing units have limited memory, which enforces a constraint on the possible components that can be deployed into each hardware host. Formally, the memory constraint is defined as

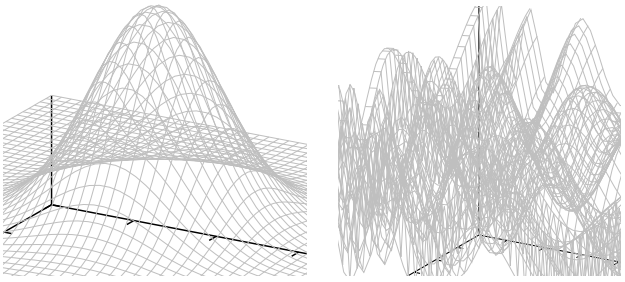
$$\sum_{i=1}^{n} \text{sz}_i x_{ij} \leq \text{cp}_j, \quad \forall j \in \{1, \ldots, m\}. \tag{9}$$

2) Communication constraint: The interaction between software components restricts their allocation, since a communication network is required if they are in different hosts. Hence, if the transition probability between two software components i and j is positive, $p_{ij} > 0$, these two components will communicate with a certain probability. Therefore, either they should be deployed on the same hardware unit, or on different units that are connected with a communication link (bus) with a positive data rate, $\text{dr}_{ij} > 0$. This is modelled as follows:

$$x_{ik} + x_{jl} \leq 1, \quad \text{if } p_{ij} > 0 \text{ and } \text{dr}_{kl} = 0. \tag{10}$$

IV. Problem Hardness

The suitability of a search method in solving an optimisation problem instance depends on the structure of the fitness landscape of that instance. A fitness landscape in the context of combinatorial optimisation problems refers to the (i) search space S, composed of all possible solutions that are connected through (ii) the distance operator, which assigns each solution $s\in S$ to a set of neighbours $N(s)\subset S$, and the fitness function $F : S\rightarrow\Re$. As the neighbourhood of a solution depends on the distance operator, a given problem can have any number of fitness landscapes. The neighbourhoods can be very large, such as the ones arising from the crossover operator of a genetic algorithm. On the other hand, a 2-opt operator creates a very small neighbourhood. The choice of the neighbourhood operator affects the structure of the fitness landscape, and as a results the performance of the optimisation algorithm.



(a) A smooth fitness landscape. (b) A rugged fitness landscape.

Fig. 1: Hypothetical fitness landscapes with different features.

Features of the fitness landscape of an optimisation problem can be characterised using characterisation metrics. Examples of features are the distribution of local optima, the topology of the global optimum, and the length of plateaus. A fitness landscape with one local optimum that is also the global optimum (e.g. fig. 1a) would be easy to search with a hill climbing method. On the other hand, when many local optima and an isolated global optimum are present (e.g. fig. 1b), the fitness landscape is rugged and hard to explore. A plateau indicates the presence of neutrality in the landscape, where the progress of a search algorithm potentially stagnates, because of the search iterating between equally fit solutions. Plateaus can be detrimental to search algorithms such as local search and hill climbing approaches, which depend on gradients in the fitness landscape they can follow. The presence of ridges, the shapes and sizes of valleys and the distance between local optima are features that decide the effectiveness of one algorithm over another.

In situations where the search algorithm is stuck in a plateau, escape mechanisms could be implemented, such as temporarily accepting non-improving moves, or restarting the search in a different position. Intuitively, characterising plateaus, such as their size or the average distance between points may help in designing better strategies for accepting non-improving moves. This information could also be used to guide the search algorithm to the global optimum, which may be isolated and surrounded by other structures, such as plateaus, valleys, hills or ridges. Usually, these structures are hard to characterise.

The choice of neighbourhood structure determines whether the fitness landscape is easy to search. For example, if the fitness difference between any two neighbouring solutions is on average small then the landscape is more likely to be suited for a wide range of local search operators. In contrast, if significant fitness difference is encountered in the neighbourhood, different operators will produce different quality results and the choice of the operator becomes important. In general, the concept of fitness landscape allows the study of the dynamics of the evolution of solutions, the comparative effectiveness of search methods and the ability of the algorithms to find good solutions to a given problem. Fitness landscape characterisation aims at understanding the relationship between the structure of the fitness landscape and the behaviour of search algorithms [33], which helps in choosing suitable methods.

Measuring the ruggedness of a fitness landscape has been one of the main areas of research in characterisation metrics. High ruggedness refers to a fitness landscape with many local optima that are distributed unevenly, and a global optimum that is hard to reach. To understand whether a neighbourhood operator creates a fitness landscape that is hard to search, fitness landscape characterisation metrics were devised. Due to computational cost of characterising the whole search space, the majority of characterisation metrics rely on samples used to infer statistical properties of the fitness landscape.

V. Component Deployment Characterisation

Predictive diagnostic optimisation (PDO) [16] has been proven useful in estimating the difficulty of the fitness space of unconstrained optimisation problems. In this paper, we adapt PDO to constrained fitness landscapes. PDO applies a local search technique to optimise solutions while attempting to predict the quality to expect at the end of the local optimisation process. To this end, predictors are created which match the profile of a certain solution. When new similar solutions are found, the closest matching predictor projects the quality to expect after locally optimising the solution. This allows for metrics such as predictor count and prediction error.

When the local optima of a problem instance's search space are very diverse, the algorithm creates a greater number of predictors, as the existing predictors often do not match the solutions within the specified margin of tolerance. It has also been observed that a higher error in the predictions made coincides with the discovery of solutions of lower quality [16]. In this work, we explore how PDO can reflect the difficulty of a search space when it arises from the constrained properties of the problem rather than the combinatorial complexity.

A. Search Space

The search space of the component deployment problem is defined as the set of candidate deployment architectures. Formally, each component deployment architecture alternative is defined as $s_i = [d(c_1), d(c_2), ..., d(c_n)]$, where $d(c_i)$ is the function $d \in D = \{d \mid d : \mathcal{C} \rightarrow \mathcal{H}\}$ that returns the hardware host where component c_i is allocated.

We investigate the 2-opt operator, which swaps the allocation of two components that are next to each other in the solution representation. For examples, given solution

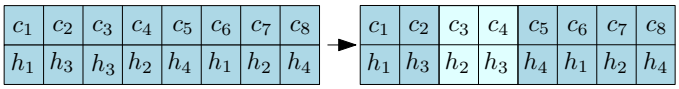


Fig. 2: The 2-opt search operator.

$[d(c_1), d(c_2), d(c_3)..., d(c_n)]$, the 2-opt operator would swap the allocation of component c_2 with c_3, as in fig. 2.

To illustrate the fitness landscape of the component deployment problem, and the effect constraints have on the progress of the search, we consider the problem of allocating three software components into two hardware hosts (h2 c3). For simplicity, only the memory constraint is considered at this stage. The settings for the parameters of the software and hardware architecture are presented in tables I and II respectively. The notations have been described in section III.

TABLE I: Parameters of the software architecture.

	Components			**Interactions**			
	sz_i	wl_i	q_i	p_{ij}	c_0	c_1	c_2
c_0	256	0.4	0.8	c_0	0.0	0.99	0.01
c_1	64	2.4	0.0	c_1	0.0	0.0	0.0
c_2	256	0.4	0.2	c_2	0.49	0.51	0.0

TABLE II: Properties of the hardware architecture.

	Hosts			**Network**	
	fr_i	cp_i	ps_i	dr_{ij}	fr_{ij}
h_0	0.004	128	100	128	0.006
h_1	2.0E-5	512	40		

The search space (all candidate solution) of h2 c3, their fitness, and feasibility are depicted in table III. The first column shows the possible allocations of the three components, where 0,0,1 means that the first and the second components (c_0, c_1) have been allocated in host 0, whereas the third component (c_2) has been allocated in host 1.

TABLE III: Search space of h2 c3.

Solution	Fitness	Feasibility
0,0,0	0.99973	false
0,0,1	0.99977	false
0,1,0	0.99996	false
0,1,1	0.99997	false
1,0,0	0.99985	false
1,0,1	0.99990	**true**
1,1,0	0.99998	false
1,1,1	0.99999	false

The visualisation of the fitness landscape is not possible since the problem has four dimensions (three components and the fitness), however, it is possible to see how this kind of fitness landscape would be hard to explore with a local search method. The example we consider is small enough to search exhaustively, but consider a large search space where feasible solutions are surrounded by infeasible areas. Local search methods follow gradients that lead to local optima.

If the memory constraint is not considered, the local search method initialised with solution 0,0,0, and using the 1-flip operator which changes the value of one solution component would follow a gradient through the following solutions: 1,0,0 (0.99985), 1,1,0 (0.99998), 1,1,1 (0.99999). The local optimum (1,1,1) is the solution with all components allocated to host

1. This produces the highest reliability, since it avoids the network, which contributes to the overall probability of system failure with its own failure rate. In addition, the failure rate of host 1 is much lower, which makes the deployment of components in this host more desirable.

However, all solutions but 1,0,1 with reliability 0.99990 are infeasible. In this example, the local search method would have to pick 1,0,1 as a starting point to be able to find it, since there are no feasible gradients in the fitness landscape that lead to it. In this example, the optimisation algorithm would in fact be outperformed by a random picking, since it wastes function evaluations in exploring the neighbourhood.

B. Predictive Diagnostic Optimisation

Predictive Diagnostic Optimisation (PDO) is an Iterated Local Search (ILS) method that records the predictability of the local search process while finding optima using a sampling-based ascent descent method. A detailed description has been published earlier by Gheorghita et al. [16]. We reiterate the crucial elements for the reader's convenience. The main steps of the method are shown in algorithm 1 and described in detail in the following subsections.

PDO applies a steepest-ascent-based local search process to a solution s_i that is randomly initialised (line 6 in algorithm 1). The 2-opt operator has a linear time complexity, hence at each step, all neighbours can be checked to select the best one. In the general case, the $k-$opt operator is linear for $k \leq 3$, and becomes NP-complete for k greater than 3. In complex neighbourhood relations, the neighbourhood of the current solution is explored until a representative sample has been reached. The change that entails the largest fitness gain is then applied to the solution and the ratio of fitness improvement to initial solution fitness $f(s_i)$ is calculated according to eq. 11.

$$p'_i = \frac{f(s_i) - f(s'_i)}{f(s_i)} \tag{11}$$

This fitness ratio is later used to match the predictor to a future solution. For a comparison, the best possible fitness improvement that can be made to a newly found solution by applying a single change is established. The predictor with the closest matching ratio of p'_i is used to predict the final fitness of the new solution. A second ratio p''_i is recorded between the total fitness improvement and the fitness improvement made by the first change. Eq. 12 illustrates this with $f(s''_i)$ denoting the fitness of the local optimum, which is achieved when the sampling cannot detect any further improvement.

$$p''_i = \frac{f(s_i) - f(s''_i)}{f(s_i) - f(s'_i)} \tag{12}$$

Every new initial solution s_j is matched to a predictor after the first improvement has been made and the ratio p'_j has been established. The closest matching predictor is used to project the expected final fitness $f_e(s''_j)$ using eq. 13.

$$f_e(s''_j) = p''_i * (f(s_j) - f(s'_j)) + f(s_j) \tag{13}$$

Although the number of changes made to a solution varies depending on the depth of the local optimum's basin, the projection of the expected fitness $f_e(s''_j)$ only uses the fitness at a distinct point in the solution's descent towards the optimum to predict the final fitness. PDO has two main stages, as shown in algorithm 1: predictor discovery and predictor application.

Algorithm 1 Predictive Diagnostic Optimisation

```
    procedure PredictorDiscovery(N, ε)
2:      predictorCount ← 0
        previousCount ← −1
4:      while predictorCount ≠ previousCount do
            for i ← 1, N do
6:              s_i ← RandomSolution
                s'_i = FirstLocalSearchStep(s_i)
8:              p'_i = (f(s_i) − f(s'_i)) / f(s_i)
                s''_i = SecondLocalSearchStep(s'_i)
10:             p''_i = (f(s_i) − f(s''_i)) / (f(s_i) − f(s'_i))
                different ← True
12:             for j ← 1, predictorCount do
                    C(p_i, p_j) = |p'_i − p'_j| / (|p'_i| + |p'_j|) + |p''_i − p''_j| / (|p''_i| + |p''_j|)
14:                 if C(p_i, p_j) > ε then
                        different ← False
16:                 end if
                end for
18:             if different then
                    SavePredictor(p_i)
20:             end if
            end for
22:         previousCount ← predictorCount
            predictorCount ← predictorCount + 1
24:     end while
    end procedure
26: procedure PredictorApplication(p)
        for i ← 1, N do
28:         s_i ← RandomSolution
            s'_i ← FirstLocalSearchStep(s_i)
30:         p'_i ← (f(s_i) − f(s'_i)) / f(s_i)
            different ← LargeDouble
32:         for p ∈ p do
                C(p_i, p) = |p'_i − p'| / (|p'_i| + |p'|) + |p''_i − p''| / (|p''_i| + |p''|)
34:             if C(p_i, p) < different then
                    p* ← p
36:             end if
            end for
38:         s''_i = SecondLocalSearchStep(s'_i)
            RecordPredictionError(s''_i, p*)
40:     end for
    end procedure
```

1) Predictor discovery: During predictor discovery, solutions to the problem instance are repeatedly created by uniform random sampling. The goal is to create the necessary predictors which represent the complete fitness landscape as accurately as possible. Therefore, whenever a new solution gives rise to creating a predictor which is significantly different from the existing ones, the new predictor is retained. This process continues until no new predictors have been created in a predefined number of attempts.

When a new solution s_i has been created and subsequently changed by a single improvement to s'_i, the ratios p'_i and p''_i are calculated according to Equation 11 and 12. PDO then calculates the Canberra distance [20] between the candidate predictor and all existing predictors to determine whether the predictor should be added to the archive of predictors.

$$C(p_i, p_j) = \frac{|p'_i - p'_j|}{|p'_i| + |p'_j|} + \frac{|p''_i - p''_j|}{|p''_i| + |p''_j|} \quad (14)$$

where $p_i = (p'_i, p''_i)$ and $p_j = (p'_j, p''_j)$ are the predictors with their respective ratios of fitness gain. $C(p_i, p_j)$ is defined in the range [0,2), with greater values representing greater difference between predictors.

Each time a new candidate predictor is created, its Canberra distance (eq. 14) to all existing predictors is calculated. If the distance is above a predefined threshold of $\epsilon = 0.1$, the predictor is added to the existing predictors. The predictor creation phase stops if no new predictors are accepted. The predictor count acts as an additional indicator to assess the information provided by the prediction error and the best fitnesses. If the predictor number is low, the quality of new predicted solutions is proportional to the level of the error rate. The higher the error rate, the less we can guarantee the quality of the predictions. If the predictor number is higher, we can have more confidence in the predicted locally optimal fitnesses despite the prediction error rates.

2) Predictor application: In the predictor application phase, only existing predictors are applied to solutions by closest match and the resulting prediction errors are recorded (line 39 in algorithm 1). The prediction error is calculated as a percentage error between the predicted and the actual fitness for each of the solutions that were optimised during this phase.

As in previous work, a predictor is selected for making predictions based on the minimum difference between its first fitness improvement and the candidate solution's first step's fitness improvement. This is the Canberra distance given in eq. 14 and line 33 of algorithm 1. The solution is then optimised to the local optimum and the real fitness is compared to the predicted fitness to obtain the difference, which comprises the prediction error.

VI. Experiments

Constraints reduce the number of feasible solutions. The effect of constraints on the performance of the algorithm, however, is not completely understood. Some constraints may create isolated feasible regions, which are unreachable from other feasible regions. In such constrained search spaces, a local search method would normally fail to explore the candidate solutions effectively.

A set of experiments were designed to investigate the constrained search space of the component deployment problem, with a particular focus on the degree of interactions between software components. The aim is to analyse the topology of the fitness landscape in the component deployment problem, and how it is shaped by the amount of interactions between software component. In essence, we seek to answer the following research questions:

RQ1: Does increasing interactions between software components create a fitness landscape where solutions with high reliability are hard to find?

RQ2: If so, why are high quality solutions hard to find?

A. Experimental Design

The problem instances used for the experiments were generated using ArcheOpterix [2], which is platform for modelling, evaluating and optimising embedded system architectures. The latest version of ArcheOpterix used in the experiments can be downloaded from http://users.monash.edu.au/~aldeidaa/ArcheOpterix.html. Problems with varying complexity and constrainedness were considered. The communication constraint is affected by the percentages of components that interact with each other. The degree of interactions was varied between 0% (no interaction), 10%, 25%, 50%, 75% and 100%. The problem instances with varying size are: h10 c23 (10 hosts, 23 components), h20 c45 (20 hosts, 45 components), h30 c65 (30 hosts, 65 components), h45 c87 (45 host, 87 components), and h60 c130 (60 hosts, 130 components). In total, 30 problem instances were created.

The training phase of PDO creates the predictors and stops when all new solutions match the existing predictors within a margin of tolerance; in our experiments, the training phase ends when no new predictors are created for 1000 function evaluations. In the testing phase, the closest-matching predictor was applied to each solution before and after the local optimisation to determine the prediction error, calculated as *sum of squared error*. The LS terminates after 20 000 function evaluations. Due to the stochastic nature of this process, 30 runs were performed for each problem instance and optimisation scheme. During both stages of diagnostic optimisation, the initial solutions for the LS were created uniformly randomly and infeasible solution were discarded.

The component deployment problem, the local search, and the fitness characterisation metrics are implemented in ArcheOpterix [2]. The problem instances used in the experiments and the complete experimental results can be downloaded from http://users.monash.edu.au/~aldeidaa/ArcheOpterix.html. The optimal solutions of the instances generated for the purpose of this experiment are not known, hence we are only able to report the reliabilities of the best solutions found as absolute values.

B. Problem Features

We cannot expect a single metric to identify the type of landscape and accurately estimate the quality to expect from a search. Therefore, in this work, we consider a number of metrics that can easily be recorded during a PDO trial.

i) **predictor count** is the number of predictors created during the training phase,
ii) **prediction error** is calculated as a percentage error between the predicted and the actual fitness for each of the solutions that were optimised during the
iii) **predictor diversity** is measured using the Canberra distance described in eq. 14,
iv) **solution diversity** is measured as the hamming distance of local optima over 30 trials,
v) **solution fitness** is the reliability of software architectures measured as described in eq. 7,
vi) **feasibility ratio** is the ratio of feasible solutions over all solutions explored during the optimisation process,
vii) **optima count** is the total number of local optima found during the optimisation phase.

C. Results

Measuring hardness of an instance is typically done by comparing the fitness of the solution reached after a certain number of iterations, or by comparing the number of iterations taken to reach the best solution compared to other algorithms. Since the problem instances considered in this paper are too large to solve exhaustively, it is not possible to find the best solutions, hence we use the best fitness reached after a fixed number of iterations.

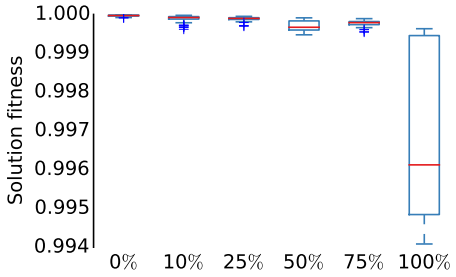


Fig. 3: Reliability values of the 30 trials and 30 different component deployment instances.

To check whether a higher degree of interactions between software components creates a fitness landscape where solutions with high reliability are hard to find, we investigate the qualities achieved by the 30 trials of the optimisation algorithm on 30 component deployment instances. The results of the optimisation process are shown as boxplots in fig 3. The means and standard deviations for the different interaction levels are shown in table IV.

TABLE IV: Means and standard deviations of the 30 trials and 30 problem instances for each interaction level.

Interactions	Mean fitness	Mean prediction error
0%	0.999968	2.1e-05
10%	0.999891	8.4e-05
25%	0.999883	4.9e-05
50%	0.999704	1.2e-04
75%	0.999773	7.8e-05
100%	0.996707	2.1e-03

The quality of the optima found is high until there are too many constraints to leave enough basins and the quality of the solutions found deteriorates remarkably. This can be clearly seen in fig. 3, and table IV, where the problem instances with the highest degree of interactions have the lowest fitness.

It appears that the tightness of constraints affects the difficulty in solving the component deployment problem. Indeed, a strong negative correlation can be observed between fitness and interaction level, as shown in fig. 4. This may mean that problem instances with higher degree of interactions create a fitness landscape that is hard to search, with good quality local optima that are isolated and difficult to reach. However, it may be possible that in very constrained search spaces better solutions no longer exist or that the LS with its feasibility preservation mechanism cannot reach them behind infeasible space. This relates to the second research question, where we are seeking to understand why in more constrained search spaces harder solutions are hard to find. To answer this question, we investigate the correlation strengths between component interactions, solution fitness, and fitness landscape

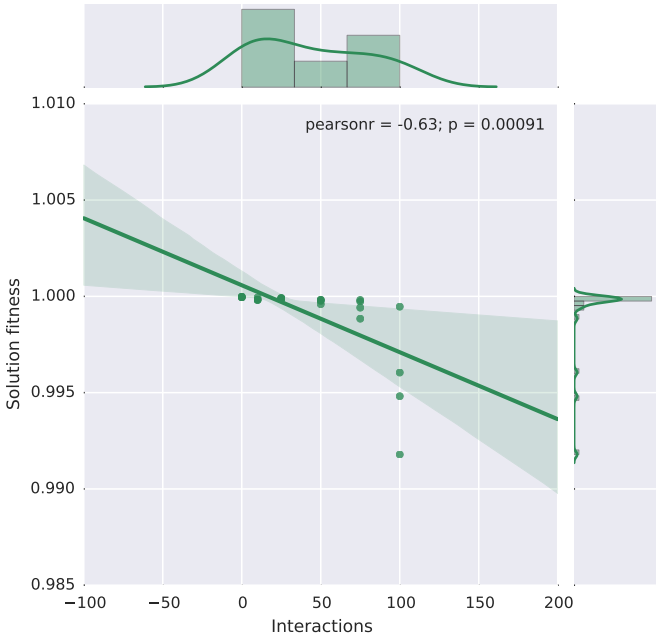


Fig. 4: Correlation between interactions and solution fitness.

characterisation metrics, namely predictor count, prediction error, predictor diversity, feasibility ratio, optima count, and solution diversity. Results are shown in fig. 5.

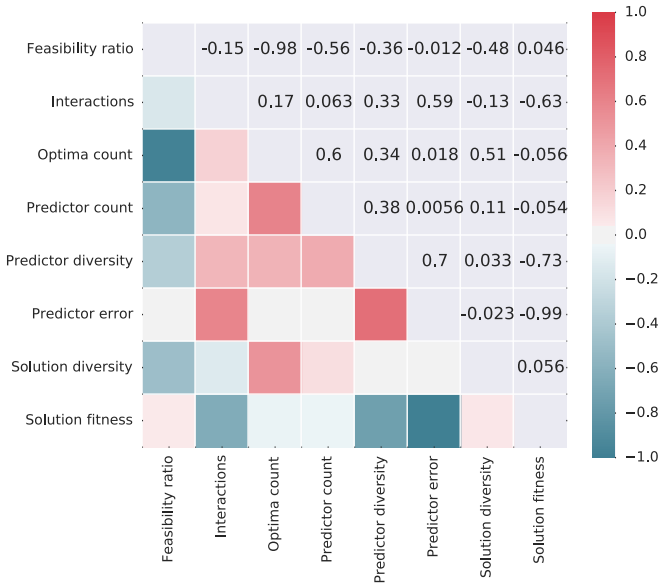


Fig. 5: The correlation strengths between problem features and fitness landscape characterisation metrics.

The feasibility ration measures the proportion of solutions encountered during the search that satisfy the constraints. Naturally, it is harder to find feasible solutions when constraints become tighter, as evidenced by the negative correlation coefficient between feasibility ratio, measured as the number of feasible solutions over all solutions explored during the optimisation phase, and tightness of interaction constraint. However, the correlation strength is weak, which indicates that feasible areas are not completely isolated.

The number of local optima increases with higher levels of interactions, as shown by the positive correlation, although not very strongly. The optima count is a measure of ruggedness, and it can be derived that the interaction constraints create more rugged fitness landscapes. However, in the more constrained fitness landscapes, the local optima are closer to each-other, as indicated by the negative correlation between solution diversity (measured using hamming distance) and interactions.

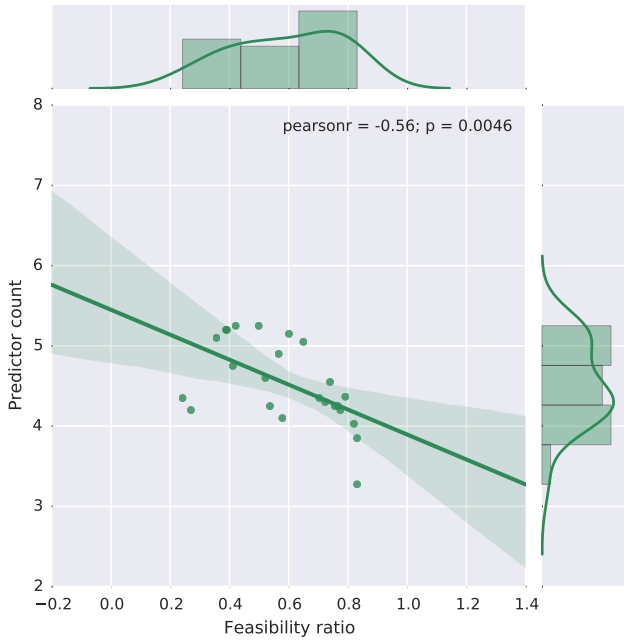


Fig. 6: Correlation between feasibility ratio & predictor count.

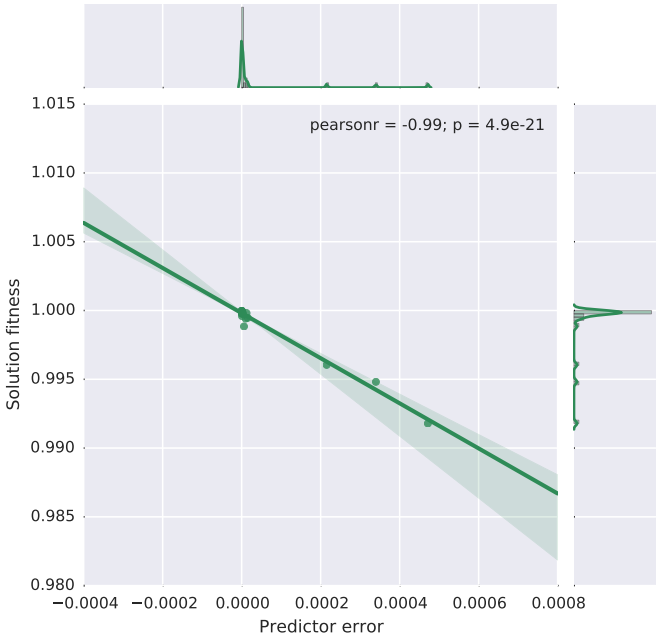


Fig. 7: Correlation between prediction error and solution quality.

As the feasible search space becomes smaller with the rising numbers of interactions between software components, the number of predictors increases, as shown in fig. 6. The predictor count depends on the similarity of the gradients that are encountered in the search space, which is an indication of the homogeneity of the basins of attraction. If the slopes to these basins with a local optimum at the top have the same shape, a single predictor will suffice to predict the outcome of a full local search. Combinatorially complex problems with rugged landscapes typically have predictors of limited accuracy, as the slopes all differ.

From all investigated metrics, prediction error correlates consistently with solution quality for all problem instances; the correlation strength is -0.99 (see fig. 7). Prediction error is a measure of the presence of patterns in the fitness landscape. The prediction error is low if the fitness landscape is composed of structures (gradients) that can be modelled during the training phase. Rugged fitness landscapes, on the other hand, would contain information that can not be compressed, or modelled with predictors. This would result in high prediction errors. It can be concluded that interaction constraints in the component deployment problem increase the ruggedness of the fitness landscape, by increasing the number of local optima, and creating more diverse gradients that are harder to predict. Hence, high quality solutions are hard to find due to high ruggedness, and not isolation of feasibility regions.

VII. Conclusion

In this work, the effect of constraints on reliability optimisation in the component deployment problem was investigated. The degree of interaction between software components changes the structure of the fitness landscape, which impacts the ability of the search algorithm to find high quality solutions. Fitness landscape characterisation was performed to estimate the ruggedness of the search space, characterise the problem instances that reflect practical deployment problems, and judge the suitability of the search operator applied.

Unlike previous applications of fitness landscape characterisation, the problem instances examined here are highly constrained. It was observed that the degree of interaction between software components affects the structure of the fitness landscape. The interaction constraint creates feasible regions with many local optima and diverse basins of attractions that prohibit the search method to explore them effectively. Higher prediction errors were observed for tighter interaction constraints, which is attributed to a rugged fitness landscape that arises from the difficult structure of the combinatorial component deployment problem.

As the levels of interaction restrictions were varied, we were able to observe that highly constrained problems have more predictors and high prediction errors in combination with more local optima. When the problem is highly constrained in terms of interacting components, the basins of attraction are diverse and the predictors created in the training phase, although in high numbers, are not representative of the entire search space, which leads to a high prediction error. When the problem is less constrained, the search becomes less complex, the fitness landscape is smoother, and the number of predictors is smaller. The number of local optima, however, was not affected greatly by the tightness of constraints.

In essence, this study shows that interaction constraints affect the structure of the fitness landscape, as evidenced by the strong correlation between interaction level and solution quality. The characterisation metrics can be used to understand the challenging features or properties of other problem instances, and investigate the behaviour of other optimisation methods in search-based software engineering. A key challenge with fitness landscape characterisation is to adequately characterize the problem instance search space by devising suitable measures, hence in the future, the investigation of other characterisation metrics for constrained fitness landscapes is a priority.

Threats to validity: Predictors are created based on a sampling process, which may lead to different results in different trials of the same problem instance. This threat was reduced by using 30 trials for each problem instance. Furthermore, the results presented in the paper may be affected by the implementation of the approach, which may contain errors or bias towards favourable results. To avoid this, we have followed regular code-review sessions in several occasions, and cross-checked the implementation and the conceptual design.

Acknowledgements: This research was supported under Australian Research Council's Discovery Projects funding scheme, project number DE 140100017.

REFERENCES

[1] A. Aleti. Designing automotive embedded systems with adaptive genetic algorithms. *Automated Software Engineering*, pages 1–42, 2014.

[2] 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*, pages 61–71. ACM and IEEE Digital Libraries, 2009.

[3] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 39(5):658–683, 2013.

[4] A. Aleti, L. Grunske, I. Meedeniya, and I. Moser. Let the ants deploy your software - an ACO based deployment optimisation strategy. In *Automated Software Engineering*, pages 505–509. IEEE Computer Society, 2009.

[5] A. Aleti and I. Meedeniya. Component deployment optimisation with bayesian learning. In *ACM Sigsoft Symposium on Component Based Software Engineering*, pages 11–20. ACM, 2011.

[6] I. Assayad, A. Girault, and H. Kalla. A bi-criteria scheduling heuristic for distributed embedded systems under reliability and real-time constraints. In *Dependable Systems and Networks*, pages 347–356. IEEE Computer Society, 2004.

[7] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transaction on Dependable Secure Computing*, 1(1):11–33, 2004.

[8] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*, volume 54. Addison-Wesley Longman Publishing Co., Inc., 2003.

[9] R. Calinescu, L. Grunske, M. Z. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qoS management and optimization in service-based systems. *IEEE Transaction on Software Engineering*, 37(3):387–409, 2011.

[10] D. W. Coit and A. Konak. Multiple weighted objectives heuristic for the redundancy allocation problem. *IEEE Transactions on Reliability*, 55(3):551–558, 2006.

[11] D. W. Corne and A. P. Reynolds. Optimisation and generalisation: Footprints in instance space. In *Parallel Problem Solving from Nature*, volume 6238 of *LNCS*, pages 22–31. Springer, 2010.

[12] M. J. Csorba, P. E. Heegaard, and P. Herrmann. Cost-efficient deployment of collaborating components. In *Distributed Applications and Interoperable Systems*, volume 5053 of *LNCS*, pages 253–268, 2008.

[13] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology*, 14(2):199–245, Apr. 2005.

[14] M. Eisenring, L. Thiele, and E. Zitzler. Conflicting Criteria in Embedded System Design. *IEEE Design and Testing*, 17(2):51–59, 2000.

[15] J. Fredriksson, K. Sandström, and MikaelÅkerholm. Optimizing resource usage in component-based real-time systems. In *Component-Based Software Engineering*, volume 3489 of *LNCS*, pages 49–65. Springer, 2005.

[16] M. Gheorghita, I. Moser, and A. Aleti. Designing and characterising fitness landscapes with various operators. In *IEEE Congress on Evolutionary Computation*, pages 2766–2772. IEEE, 2013.

[17] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45(1):11:1–11:61, Dec. 2012.

[18] ISO/IEC. IEEE International Standard 1471 2000 - Systems and software engineering - Recommended practice for architectural description of software-intensive systems, 2000.

[19] P. Kubat. Assessing reliability of modular software. *Operations Research Letters*, 8(1):35–41, 1989.

[20] G. Lance and T. Williams. Computer programs for hierarchical polythetic classification ('similarity analyses'). *The Computer Journal*, 9(1):60–64, 1966.

[21] M. R. Lyu. *Handbook of software reliability engineering*. IEEE Computer Society Press and McGraw-Hill Book Company, 1996.

[22] S. Malek, N. Medvidovic, and M. Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Transactions on Software Engineering*, 38(1):73–100, 2012.

[23] A. Martens and H. Koziolek. Automatic, model-based software performance improvement for component-based software designs. *Electronic Notes in Theoretical Computer Science*, 253(1):77–93, 2009.

[24] I. Meedeniya, B. Buhnova, A. Aleti, and L. Grunske. Architecture-Driven Reliability and Energy Optimization for Complex Embedded Systems. In *Research into Practice - Reality and Gaps, Quality of Software Architectures*, pages 52–67. Springer, 2010.

[25] I. Meedeniya, B. Buhnova, A. Aleti, and L. Grunske. Reliability-Driven Deployment Optimization for Embedded Systems. *Journal of Systems and Software*, 84(5):835–846, 2011.

[26] I. Moser and J. Montgomery. Population-ACO for the automotive deployment problem. In *Genetic and Evolutionary Computation*, pages 777–784. ACM, 2011.

[27] I. Moser and S. Mostaghim. The automotive deployment problem: A practical application for constrained multiobjective evolutionary optimisation. In *Evolutionary Computation, IEEE Congress on*, pages 1–8, 2010.

[28] Y. Papadopoulos and C. Grante. Evolving car designs using model-based automated safety analysis and optimisation techniques. *The Journal of Systems and Software*, 76(1):77–89, 2005.

[29] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transaction Computers*, 55(2):99–112, 2006.

[30] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *Future of Software Engineering*, pages 55–71. IEEE Computer Society, 2007.

[31] S. Shan and G. G. Wang. Reliable design space and complete single-loop reliability-based design optimization. *Reliability Engineering and System Safety*, 93(8):1218 – 1230, 2008.

[32] K. Smith-Miles and L. Lopes. Measuring instance difficulty for combinatorial optimization problems. *Computers and Operations Research*, 39(5):875 – 889, 2012.

[33] P. Stadler. Fitness landscapes. *Applied Mathematics and Computation*, 117:187–207, 2002.

[34] K. Trivedi. *Probability & Statistics with Reliability, Queuing and Computer Science Applications*. Wiley-India, 2009.

[35] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.