Automated Software Engineering manuscript No.
(will be inserted by the editor)

Designing Automotive Embedded Systems with Adaptive Genetic Algorithms

Aldeida Aleti

the date of receipt and acceptance should be inserted later

Abstract One of the most common problems faced by planners, whether in industry or government, is optimisation - finding the optimal solution to a problem. Even a one percent improvement in a solution can make a difference of millions of dollars in some cases. Traditionally optimisation problems are solved by analytic means or exact optimisation methods. Today, however, many optimisation problems in the design of embedded architectures involve complex combinatorial systems that make such traditional approaches unsuitable or intractable. Genetic Algorithms (GAs), instead, tackle these kind of problems by finding good solutions in a reasonable amount of time. Their successful application, however, relies on algorithm parameters which are problem dependent, and usually even depend on the problem instance at hand. To address this issue, we propose an adaptive parameter control method for Genetic Algorithms, which adjusts parameters during the optimisation process. The central aim of this work is to assist practitioners in solving complex combinatorial optimisation problems by adapting the optimisation strategy to the problem being solved. We present a case study from the automotive industry, which shows the efficiency and applicability of the proposed adaptive optimisation approach. The experimental evaluation indicates that the proposed approach outperforms optimisation methods with pre-tuned parameter values and three prominent adaptive parameter control techniques.

Keywords Software architecture · Genetic Algorithms · adaptive parameter control · Bayesian Networks

1 Introduction

The design of embedded systems, and in particular of automotive embedded systems involves several important decisions, such as which software components to select and how to deploy them into the hardware architecture. Today more than 80 percent of innovations in a car comes from software systems. Software adds distinguishing features to car models and allows hardware to be reused. With the increasing number of functions performed by software, the automotive embedded systems are becoming more complex with many design options to choose from. For instance, a simple power train control application has 3488 possible component realisations by instantiating different algorithms and their variants (Butts et al., 2001). A typical car consists of around 80 electronic fittings. Simple 'yes, no' decisions for each function yield approximately 2^{80} variants to be ordered and produced for a car. This large number of variants has to be handled technically (Butts et al., 2001), a task that can take years for the system designer. Most importantly, every single decision may affect the quality attributes of the system, such as safety, security, availability, reliability, maintainability, performance and temporal correctness (Pretschner et al., 2007; Fredriksson et al., 2005; Malek et al., 2005), which often are conflicting with each other.

To handle the complexity of this task, recent research has focused on architecture optimisation techniques, which partially automate the design of embedded architectures (Broy, 2006; Pretschner et al., 2007). The field is broadly known as Search Based Software Engineering (SBSE) (Harman, 2007a). SBSE

Aldeida Aleti
Faculty of Information Technology
Monash University, Clayton, VIC 3800, Australia
E-mail: aldeida.aleti@monash.edu

is applicable to many stages of the development of software systems, such as requirements engineering (Zhang et al., 2008), predictive modelling (Afzal and Torkar, 2011), design (Aleti et al., 2013), program comprehension (Harman, 2007b), and software testing (McMinn, 2004; Mantere and Alander, 2005). A comprehensive survey of SBSE by Harman et al. (2012b) indicates that there is an increase in research publications in the area of search based methods for Software Engineering problems. A systematic literature review in optimisation algorithms for the design of software architectures (Aleti et al., 2013) found that the majority of the approaches use Genetic Algorithms. These algorithms provide approximate results where exact approaches cannot be devised and optimal solutions are hard to find. Genetic Algorithms can be used by practitioners in a fairly 'black box' way without mastering advanced theory, whereas more sophisticated combinatorial algorithms are tailored to the specific mathematical structure of the problem at hand, and can become completely inapplicable if small aspects of the problem change. Another advantage of Genetic Algorithms is the stochastic element which helps them get out of local optima.

In recent years, it has been acknowledged that the robustness of Genetic Algorithms is mostly due to their numerous parameters such as crossover probabilities and population size (Eiben and Smit, 2011; Eiben et al., 2007). Algorithm parameters determine the search strategy for exploring the space of possible solutions, and make the optimisation procedure flexible and efficient regardless of the search space difficulty. However, poor algorithm parameterisation hinders the discovery of good solutions (Eiben and Smit, 2011; Eiben et al., 2007). The parameters required for optimal algorithm performance are known to be problem-specific, often even specific to the problem instance at hand (Eiben and Schut, 2008). Pressed for time, practitioners tend to choose parameter values based on few preliminary trials, in which various parameter settings are explored in an attempt to fine-tune the algorithm to a particular problem (Eiben and Smit, 2011). Depending on the number of parameters and their plausible value ranges, investigative trials for parameter optimisations can themselves be attempts to solve a combinatorially complex problem.

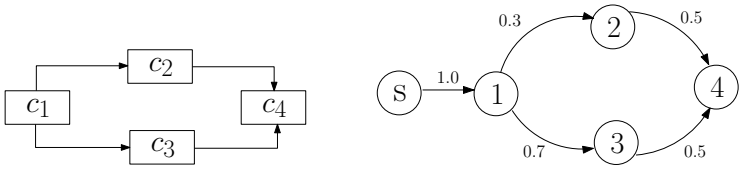
As prior tuning of algorithm parameters does not provide optimal performance, we introduce a novel and efficient adaptive Genetic Algorithm for Designing Embedded Architectures, which eliminates the requirement for tuning algorithms to particular problems. The new method adjusts algorithm parameters during the optimisation process using an adaptive parameter control strategy. The intuitive motivation comes from the way the optimisation process unfolds from a more diffused global search, requiring parameter values responsible for the exploration of the search space, to a more focused local optimisation process, requiring parameter values which help with the convergence of the algorithm. The adaptive Genetic Algorithm redefines parameter values repeatedly based on a learning process that receives its feedback from the optimisation algorithm. The method uses an evaluation of the search space to discover problem features and determine how well certain algorithm parameters perform.

The adaptive parameter control method introduced in this paper considers the stochastic nature of GAs, which may return different results for the same parameter settings. A Bayesian Network is used to model the causal relationship between parameter values and the performance of the GA. The applicability and efficiency of the proposed approach is demonstrated with the help of a real-world application from the automotive industry and a set of experiments on generated problem instances.

2 Architecture Design and Optimisation

The architecture of an embedded system represents a model or an abstraction of the real elements of the system and their properties, such as software components, hardware units, interactions of software components, and communications between hardware units. Formally, we define the software elements as a set of software components, denoted as $\mathcal{C} = \{c_1, c_2, ..., c_n\}$, where $n \in \mathbb{N}$ represents their number. The software components, illustrated in Figure 1a are considered as black boxes (Jhumka et al., 2002); unmodifiable and with unknown internal structure, but with a description of externally visible parameters. The software components interact to implement a set of *services*, which define the functional units accessed by the user of the system.

Each *service* is initiated in one software component (with a given probability), and during its execution may use many other components (possibly shared with other services), connected via communication links. For each service, the links are assigned with a transition probability. This view follows the Kubat model (Kubat, 1989), in which the software architecture is modelled as a Discrete-Time Markov Chain



(a) Software components and interactions.

(b) DTMC representing the software components and interactions.

Fig. 1: Software components and interactions represented as a DTMC.

(DTMC), where vertices represent the execution of software components, and arcs enumerate the probability of transferring the execution flow to the next component. The DTMC model is used to calculate specific quality attributes of the system, such as reliability. The model depicted in Figure 1b is constructed using the software elements shown in Figure 1a.

The hardware architecture is composed of a distributed set of hardware nodes, denoted as $\mathcal{H} = \{h_1, h_2, ..., h_m\}$, where $m \in \mathbb{N}$, with different capacities of memory, processing power, access to sensors and other peripherals. The hardware hosts are connected via network links, denoted as $\mathcal{N} = \{n_1, n_2, ...n_s\}$. An example of three hardware hosts and the network that is used for communication is depicted in Figure 2.

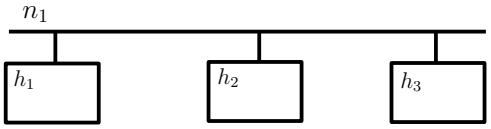


Fig. 2: Hardware elements and communication network.

Given the set of architectural elements, a series of decisions have to be taken in order to obtain the final architecture of the embedded system, such as how to deploy the software components to the hardware resources, known as the Component Deployment Problem (CDP) (Aleti and Moser, 2011), and how to assign redundancy levels to safety-critical components, known as the Redundancy Allocation Problem (RAP). Current research (Fredriksson et al., 2005; Malek et al., 2005) shows that these decisions have significant implications for the likelihood that the system achieves the desired quality attributes. In general, quality attributes are non-functional characteristics of a system, the combination of which constitutes the overall quality of the system as defined by the IEEE 1061 standard (ISO/IEC, 2000). Examples of quality attributes are reliability, safety, availability, cost, energy consumption and performance. For every design decision, several quality attributes are considered, which due to their conflicting nature, are optimised simultaneously.

2.1 Redundancy Allocation Optimisation

Redundancy Allocation (RA) determines the number of redundancies that have to be implemented for each software components. RA is one of the well-known system reliability improvement techniques (Liang and Smith, 1999; Coit and Smith, 1996; Kulturel-Konak et al., 2008). Reliability is one of the crucial aspects that should be considered when designing embedded architectures of dependable, safety critical systems such as in the automotive domain (Åkerholm et al., 2004). However, employing redundant components can have a negative influence in the other quality attributes.

The Redundancy Allocation Problem (RAP) has shown to be NP-hard (Chern, 1992), hence most of the approaches in the literature use stochastic algorithms to optimise this problem. Coit et al. (Coit and Smith, 1996) solve the Redundancy Allocation Problem (RAP) using functionally similar components, such that if one component fails, the redundant part performs the required functionality without a system failure. The optimisation problem is defined as the minimisation of cost using a genetic algorithm while satisfying a user defined system reliability level, which is handled as a constraint. Problems with hard constraints may be hard to solve with approximate methods.

To deal with this problem, Kulturel-Konak et al. (Kulturel-Konak and Coit, 2003) employ a relaxation mechanism by incorporating a penalty function into a Tabu Search. Ant Colony Optimisation with an elitist strategy which preserves the best solutions has successfully been used to solve the redundancy allocation problem (Liang and Smith, 1999). The ACO was enhanced with a mutation operator, usually not part of the ACO, to search in unexplored areas of the search space. All these approaches model the redundancy allocation problem as a singleobjective optimisation problem, usually by minimising cost while satisfying a predefined reliability criteria. Similar to Grunske (2006), who simultaneously maximises

reliability while minimising weight, we formulate the redundancy allocation problem as a multiobjective problem.

We employ the *hot spare* design topology, in which all component redundancies are active at the same time, mimicking the execution of the original component. With the N-tuple Modular Redundancy (NMR) extension the system employs a decision mechanism in the form of majority voting when multiple replicas deliver their results to the entry gate of a component. In this configuration, each component with its parallel replicas is regarded as a *subsystem*.

Formally, an architecture alternative for the redundancy allocation problem is denoted as ra. The set of all redundancy allocation candidates ra is denoted as $RA = \{ra \mid ra : \mathcal{C} \rightarrow \mathcal{N}_{RA}\}$, where $\mathcal{N}_{RA} = \{n \mid 0 \leq n \leq nMax, n \in \mathbb{N}_0\}$ delimits the redundancy level of a component. Note that, since $\mathcal{C}$ and $\mathcal{N}$ are finite, RA is also finite. An example of an architecture alternative for the Redundancy Allocation Problem is depicted in Figure 3.

The redundancy allocation problem is optimised using a Genetic Algorithm with a specialised solution encoding which maps a redundancy level to each software component. This problem has many quality-related aspects and is therefore modelled as a multiobjective problem. Each solution is encoded as $ra_i = [ra_i(c_1), ra_i(c_2), ..., ra_i(c_n)]$, where $ra_i(c_j)$ represents the redundancy level for component c_j in the architecture alternative ra_i.

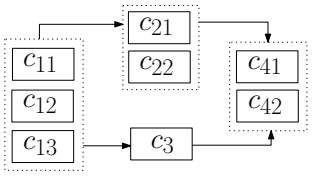


Fig. 3: Software redundancies allocated to safety critical components.

The crossover and mutation operators are used to find new architecture alternatives for RAP. The crossover operator creates two new solutions $ra'_i, ra'_j \in R$ from two parents $ra_i = [ra_i(c_1), ra_i(c_2), ..., ra_i(c_n)]$ and $ra_j = [ra_j(c_1), ra_j(c_2), ..., ra_j(c_n)]$ coming from existing population by recombining the redundancy levels, i.e. for a randomly selected index k: $ra'_i = [ra_i(c_1), ..., ra_i(c_{k-1}), ra_j(c_k), ..., ra_j(c_n)]$ and $ra'_j = [ra_j(c_1), ..., ra_j(c_{k-1}), ra_i(c_k), ..., ra_j(c_n)]$. Similarly, a single-point mutation produces a new solution ra'_i from existing ra_i by switching the redundancy levels of two software components, i.e. for randomly selected k, l: $ra'_i = [ra_j(c_1), ..., ra_j(c_l), ..., ra_j(c_k), ..., ra_j(c_n)]$ while the original solution is $ra_i = [ra_i(c_1), ..., ra_j(c_k), ..., ra_j(c_l), ..., ra_j(c_n)]$.

2.2 Component Deployment Optimisation

The Component Deployment Problem (CDP) refers to the allocation of software components to the hardware nodes, and the assignment of inter-component communications to network links. The way the components are deployed affects many aspects of the final system, such as the processing speed of the software components, how much hardware is used or the reliability of the execution of different functionalities (Meedeniya et al., 2011; Aleti et al., 2009b; Papadopoulos and Grante, 2005; Mikic-Rakic et al., 2004; Medvidovic and Malek, 2007; Malek, 2007; Sharma et al., 2005; Fredriksson et al., 2005).

Some approaches focus on the satisfaction of the constraints or user requirements (Calinescu and Kwiatkowska, 2009; Martens and Koziolek, 2009), whereas others aim at finding optimal deployments or at least candidates that are near-optimal (Medvidovic and Malek, 2007; Sharma et al., 2005; Fredriksson et al., 2005), often in combination with the given constraints (Mikic-Rakic et al., 2004; Medvidovic and Malek, 2007; Fredriksson et al., 2005; Lukasiewycz et al., 2008). In our approach, quality attributes are optimised simultaneously due to their conflicting nature. In addition, our approach considers constraints, which are incorporated into the framework. From an optimisation perspective, CDP is similar to the generalized Quadratic Assignment Problem (gQAP), where there is no restriction that one location can accommodate only a single equipment. 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 possible solution for the component deployment problem of the software and hardware architectures introduced in the previous section is depicted in Figure 4.

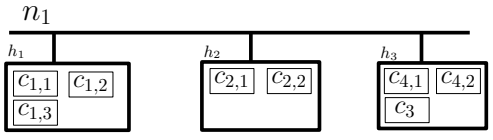


Fig. 4: Component Deployment Problem.

The component deployment problem uses a Genetic Algorithm with a specialised solution encoding which maps software components to hardware resources. Each architecture alternative of the component deployment problem is encoded for the optimisation process as $d_i = [d_i(c_1), d_i(c_2), ..., d_i(c_n)]$, where $d_i(c_j)$ represents the hardware host used to deploy component c_j in the deployment alternative d_i.

The genetic operators are applied to this representation of the solutions in order to generate new architecture alternatives. The crossover operator combines the allocation lists of two solutions. Formally, crossover creates two new solutions $d'_i, d'_j \in D$ from two parents $d_i = [d_i(c_1), d_i(c_2), ..., d_i(c_n)]$ and $d_j = [d_i(c_1), d_i(c_2), ..., d_i(c_n)]$ coming from the existing population of solutions by recombining the allocation of components, i.e. for a random k: $d'_i = [d_i(c_1), d_i(c_2), ..., d_i(c_{k-1}), d_j(c_k), ..., d_j(c_n)]$ and $d'_j = [d_j(c_1), d_j(c_2), ..., d_j(c_{k-1}), d_i(c_k), ..., d_i(c_n)]$. The mutation operator exchanges the host allocations of two randomly chosen components. Formally, mutation produces a new solution d'_i from existing d_i by switching the mapping of two components, e.g. for randomly selected k, l: $d'_i = [d_i(c_1), d_i(c_2), ..., d_i(c_k)..., d_i(c_l), ..., d_i(c_n)]$ while the original solution is $d_i = [d_i(c_1), d_i(c_2), ..., d_i(c_l)..., d_i(c_k), ..., d_i(c_n)]$. The problem definition does not allow for duplications, and a repair operation follows the crossover/mutation move.

3 Genetic Algorithms

Genetic Algorithms (GAs) (Holland, 1975) are nature-inspired optimisation methods used for solving NP-hard optimisation problems. GAs maintain a population of solutions $S = \{s_1...s_n\}$ that evolves by means of the genetic operators: the variation procedure ($\mathcal{V}$), which is composed of the mutation operator ($\hat{m}$) and the crossover operator ($\hat{c}$), the selection procedure ($\mathcal{S}$), and the replacement procedure ($\mathcal{R}$), which may also be referred to as survivor selection procedure (Eiben and Smit, 2011). The objective is to search the solution space in order to optimise the quality functions $F : S \to \mathbb{R}$.

The main steps of a GA are given in Figure 5. The optimisation process starts with a set of solutions S_0 as initial population, which can be randomly generated, created by applying heuristic rules (e.g. greedy algorithm), or provided by an expert. After the initialisation, GA evolves the population using the crossover, mutation and selection operators. The crossover and mutation operators are applied according to predefined crossover and mutation rates $\hat{c}_r, \hat{m}_r$ respectively. These operators are applied to specific solutions, which are selected by the selection procedures $\mathcal{S}$. The new individuals created by the genetic operators are added to the population. The number of offspring generated at every iteration is denoted as λ. The replacement procedure $\mathcal{R}$ selects the solutions that will survive in the next generation.

Fig. 5: The main steps of Genetic Algorithm.

```
1: procedure GA(F, μ, λ, R , V, S , m̂ , m̂_r, ĉ, ĉ_r,
   stoppingCriteria)
2:     S_0 ← RANDOM(μ))
3:     t = 0
4:     while stoppingCriteria ≠ true do
5:         for i ← 1, λ do
6:             EVALUATE(F, s_i)
7:         end for
8:         S_t(parents) = S(S_t)
9:         S'_{t+1} = V(S_t(parents), λ, m̂, m̂_r, ĉ, ĉ_r)
10:        S_{t+1} = R(S'_{t+1})
11:        t = t + 1
12:    end while
13:    RETURN(S)
14: end procedure
```

At every iteration of a GA run, the chosen parameter values are employed to create different solutions, which are evaluated using the fitness function(s). The output from the evaluation process provides valuable information, which can be used to assess the effect of the parameter values on the performance of the optimisation algorithm. We use the feedback from the search to approximate the cause of the change in the quality of the solutions and assign a quality to parameter values, denoted as $q(v_{ij})$. The quality of the parameter values is used to select the next parameter assignments, which are employed to create the next generation of solution(s).

4 Parameter Control for Genetic Algorithms

Parameter control aims at finding optimal parameter configurations for Genetic Algorithms. It describes a process where the optimisation process starts with suboptimal parameter values, which are adapted for better algorithm performance. A variety of parameter control methods exist in the literature (Eiben et al., 2007; Smith and Fogarty, 1996; Angeline, 1995; Davis, 1989; Tuson and Ross, 1998; Smith and Fogarty, 1996). Following the classification proposed by Eiben et al. (Eiben et al., 2007), parameter control methods can be grouped into three categories: deterministic parameter control, self-adaptive parameter control and adaptive parameter control.

Deterministic parameter control methods adjust parameter values based on user-defined deterministic rules, set a priori based on time, similar to the way the cooling schedule is applied in simulating annealing. Despite the promising results in the early works (Hesser and Manner, 1991; Fogarty, 1989), one of the

main issues with deterministic parameter control is setting the right schedule for changing the parameter values, since it is not obvious how to estimate the number of steps the GA will take to converge.

Self-adaptive parameter control approaches (Bäck, 1996; Farmani and Wright, 2003; Deb and Beyer, 2001; Bäck, 2001; Nadi and Khader, 2011) include the parameters to be adapted into the encoding of the individuals and evolve them simultaneously with the solutions to the problems. High-fitness solutions will survive to the next generation and propagate their superior traits together with the parameter values which are assumed to be responsible for the high quality. An inherent problem of self-adaptive parameter control methods is the increase in the size of the search space, since the search for optimal solutions has to incorporate the search for optimal parameter values, increasing the complexity of the optimisation problem.

Adaptive parameter control (Thierens, 2005; Fialho et al., 2009; Corne et al., 2002; Davis, 1989; Hong et al., 2000; Igel and Kreutz, 2003; Julstrom, 1995; Schlierkamp-Voosen and Mühlenbein, 1994; Tuson and Ross, 1998) monitor the behaviour of a GA run, which is used to adapt the parameter values, so that successful parameter values are propagated to the next generation. The update mechanism which controls the parameter values is devised a priori rather than being part of the optimisation cycle. This method does not use a predefined schedule and does not extend the solution size, which makes it a more effective way for controlling parameter values during the optimisation process.

Most of adaptive parameter control methods found in the literature (Corne et al., 2002; Davis, 1989; Hong et al., 2000; Igel and Kreutz, 2003; Julstrom, 1995) belong to the class of probability matching techniques, in which the probability of applying a parameter value is proportional to the quality of that parameter value. The earliest approaches (Igel and Kreutz, 2003; Hong et al., 2000) lacked the exploration of parameter values if the feedback from the algorithm was not in their favour in the initial phases of the process. In later work, a minimum selection probability p_{min} is introduced (Igel and Kreutz, 2003), to ensure that under-performing parameter values did not disappear during the optimisation, in case they were beneficial in the later stages of the search. These probability matching techniques select parameter values in proportion to their previous performances. One of the more recent and mature examples of probability matching is the work by Igel and Kreutz (Igel and Kreutz, 2003). Their Equation 1 for calculating the selection probability for each parameter value incorporates the maintenance of a minimum probability of selection.

$$p'_t(\upsilon_{ij}) = p_{min} + (1 - mp_{min}) \frac{p_t(\upsilon_{ij})}{\sum_{r=1}^{m} p_t(\upsilon_{ir})} \tag{1}$$

where m is the number of possible values for parameter υ_i. Probability matching has been criticised for the loose correlation between the reward allocations and the differences in performance with vastly superior values receiving a marginal increase in selection probability.

Adaptive Pursuit (AP) (Thierens, 2005) was conceived as an attempt to address this issue, ensuring that clearly superior values are rewarded with a maximum probability of choice. Even though every parameter value is selected from time to time, in practice the Adaptive Pursuit algorithm spends a number of iterations before responding to a change of best parameter value.

Dynamic Multi-Armed Bandit (DMAB) (Fialho et al., 2009) addresses the problem by completely recalculating the probabilities when a change in the effects distribution is detected by using a change detection test, in this case the statistical Page-Hinkley (PH) test. The PH test checks whether the quality of the parameter values has changed. When a change is detected, the algorithm is restarted. As a result, DMAB can quickly identify the new best parameter value without being slowed down by old information.

Adaptive Range Parameter Selection (ARPS) (Aleti et al., 2012) is the first attempt at adapting real-valued parameter ranges. The ARPS algorithm discretises continuous-valued parameters by partitioning its ranges into two equal intervals. At every iteration, the best-performing interval is subdivided by splitting it in the middle. At the same time, the worst-performing interval is merged with the worse-performing of its neighbours. The selection probabilities of the split intervals are initially the same, the merged interval maintains the neighbour's probability of selection.

Refining the most successful parameter areas into narrower intervals, the probability of choosing particular values increases. Merging badly performing parameters decreases their probability of selection. An analysis of the mutation/crossover ranges (Aleti et al., 2012) revealed that ARPS sometimes absorbs high-performing intervals into very large ranges as a result of short-term underperformance. The merged ranges are usually very large and it takes many iterations for an absorbed sub-range to re-establish itself. This behaviour almost certainly affects the performance of the search. An improved approach to

maintaining parameter value intervals uses an entropy-based measure for discretising parameter values into successful and unsuccessful ranges.

State-of-the-art parameter control methods assume that the improvement in the quality of the solutions is directly related to the use of certain parameter values. For instance, if a mutation rate of 0.01 produces a solution which improves the quality of its parent by Δq, the quality improvement Δq is considered as the effect of this mutation rate. If more than one solution is created, the sum or average of the quality improvement of all solutions is assigned as the overall effect. However, the performance of Genetic Algorithms is affected by more than one parameter value, hence using the performance of the algorithm directly as an indication of parameter success may be misleading. Moreover, Genetic Algorithms are stochastic systems, which may produce different results for the same parameter values (DeJong, 2007). Ideally, the randomness induced by the stochastic behaviour of GAs should be taken care of by the parameter control strategy. In this paper, we introduce a new method that deals with this issue.

5 Bayesian Parameter Control for Genetic Algorithms

In our approach, the optimisation process is carried out in $X = \{x_1, x_2, ..., x_k\}$, k independent algorithm instances. An algorithm instance can have one or more solutions depending on the optimisation problem at hand. Each algorithm instance evolves independently and every iteration reports to an algorithm manager, which coordinates all algorithm instances. At the start of the optimisation process, the algorithm manager randomly assigns each instance different parameter values. However, the same parameter value can be assigned to more than one instance. Figure 6 illustrates this process for n parameters ($\{\upsilon_1, ..., \upsilon_n\}$).

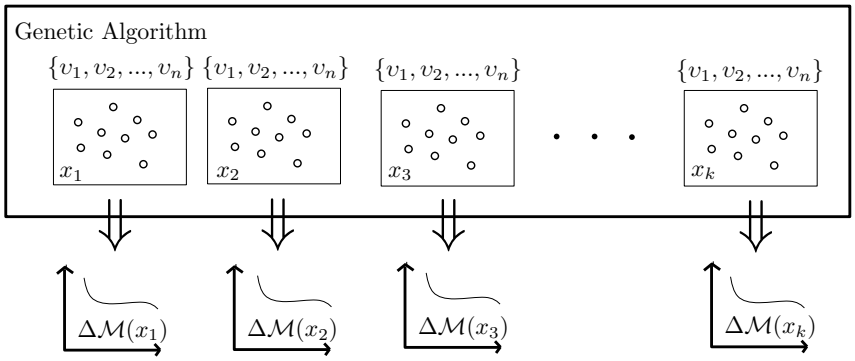


Fig. 6: Assessing the effect of parameters with parallel instances of the optimisation algorithm

Each algorithm instance has the same number of solutions, in the range $[1, \mu]$, which can also be subject to optimisation. During every iteration, new solutions are created in each algorithm instance by using the parameter values assigned to them. At the end of each generation, i.e. after the required number of offspring has been generated, the algorithm instances report the performance achieved during that generation to the algorithm manager. We denote the set of performance metrics used to measure the performance of a Genetic Algorithm as $\mathcal{M} = \{m_1, m_2, ..., m_p\}$. At each iteration, the achieved performance for each algorithm instance is measured by calculating the improvement in the properties of the current solutions, measured by these metrics, with respect to the properties of the solutions in the previous iteration, as $\Delta\mathcal{M}(x_i) = \mathcal{M}(x_i^t) - \mathcal{M}(x_i^{t-1})$.

The effect of parameter values on this performance change is denoted as e. This performance difference is used to determine whether the parameter value was successful in the previous iteration, denoted as e^+, or unsuccessful, denoted as $e = e^-$. Success is defined as producing solutions with performance values above a certain threshold th. Given an algorithm instance x_i, the effect of parameter values used in that instance is calculated as:

$$e(x_i) = \begin{cases} e^+ & \text{if} \quad \Delta\mathcal{M}(x_i) > th \\ e^- & otherwise \end{cases} \tag{2}$$

If the difference between current (time t) $\mathcal{M}(x_i^t)$ and the value of the performance metric at the previous iteration $\mathcal{M}(x_i^{t-1})$ is above th, the instance is deemed successful ($e(x_i) = e^+$), which also means

that the parameter values used to produce the solutions are considered successful. The value of the threshold th determines the greediness of the algorithm. For smaller values of th (e.g. 5% of the best solutions) the focus of parameter control is narrowed to the absolute best-performing parameter values. Larger values of th (e.g. 50% of the best solutions) allow a better exploration of parameter values with a moderate performance. Different values of the threshold were experimentally evaluated for different problem instances. It was observed that using the median as threshold produces optimal results for all problems instances investigated

In this work we use Bayesian Networks (BNs) (Pearl, 1988) as a probabilistic parameter effect assessment strategy. BNs measure cause-effect relationships between variables in a probabilistic way. They are considered to be effective tools in modelling systems which do not have complete information about the variables and the available data is stochastic or not completely available, which makes them suitable for our application.

BNs are represented as directed acyclic graphs (DAGs) with probability labels for each node as a representation of the probabilistic knowledge. The graph is defined as a triplet (V, L, P). $V = (v_1, v_2, ...v_n)$ is a set of n variables represented by nodes of the DAG. We think of each variable as an event which consists of a finite set of mutually exclusive states. It is also possible to represent continuous variables, representing numerical values, such as crossover probabilities by discretising them into a number of intervals. L is the set of links that denote the causal relationship among the variables V, modelled as directed arcs between nodes of the DAG. The directed links between variables represent dependence relationships, as shown in Figure 7a. A link from a variable v_1 to variable v_2 indicates that v_2 depends on the parent node v_1, which is denoted as $\pi(v_2)$. It can also be explained as a causal relationship where v_1 is the cause of the event v_2. The lack of a link between two nodes means that the nodes are not dependent on each other.

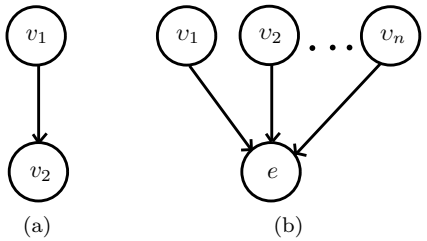


Fig. 7: Bayesian Networks.

We use BNs to model the relationship between algorithm parameters and their effect on the performance of the algorithm. The effect on the algorithm is represented by a child node (e), which can take two values: e^+ for successful and e^- for unsuccessful. The child node e has many parent nodes, which represent the algorithm parameters, denoted as $\{v_1, v_2, ..., v_n\}$, as shown in Figure 7b. Each node v_i can take different values, which we denote similarly to parameter values (ranges) as $\{v_{i1}, ..., v_{im_i}\}$. Each of the parent nodes v_i is annotated with probabilities $p(v_{ij})$, as shown in Figure 8. These represent the prior probabilities, which can be calculated as classical probabilities, to represent the probability that a certain parameter value will occur, or as Bayesian probabilities, to represent a belief that a certain parameter value will be used. In the first iteration, we assign equal prior probabilities to each value of the parameters, since they all have the same probability of being selected, calculated as $p(v_{ij}) = 1/m_i$, where m_i is the total number of values or intervals of parameter v_i. This represents our belief in how frequently a certain parameter value should be used. These prior probabilities are used to select the parameter values to assign to the algorithm instances in the first iteration. The probability that the child node e attains a certain value, in our case e^+ or e^-, depends on the probabilities of the parent nodes, i.e. the probabilities of the parameter values. This relationship is measured as a conditional probability, i.e. the probability of an event occurring given that another event has already happened, defined as: $P = \{p(e \mid \pi(e)) \mid \pi(e) \in V\}$, where $\pi(e)$ are the parents of node v.

To calculate the conditional probabilities, every iteration, we establish the number of times the value has been used, denoted as $n(v_{ij})$ and the number of times the value led to a success $n(v_{ij} \wedge e^+)$ (the number of successful instances that use that parameter value). Then we calculate the conditional probabilities for each parameter value by using Equation 3. These conditional probabilities represent the success rate of each parameter value in the current iteration.

$$p(e^+|v_{ij}) = \frac{n(v_{ij} \wedge e^+)}{n(v_{ij})} \tag{3}$$

where $n(v_{ij} \wedge e^+)$ is the successful attempts that use v_{ij} and $n(v_{ij})$ is the number of times v_{ij} is used. The conditional probabilities of all parameter values $v_{ij}, i \in n, j \in m$ are calculated for all parameter values resulting in the annotated graph shown in Figure 8.

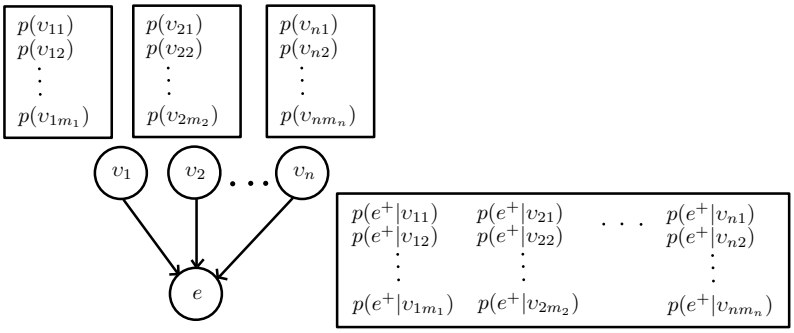


Fig. 8: A Bayesian Network annotated with conditional probabilities employed by the parameter effect assessment strategy.

Fig. 9: Bayesian Parameter Control Strategy

```
1: procedure BayesianParameterControl
2:     for all parameter v_i, i ∈ n do
3:         for all algorithm instance x_s, s ∈ k do
4:             for all parameter value v_ij, j ∈ m do
5:                 if v_ij is used in x_s then
6:                     n(v_ij) = n(v_ij) + 1
7:                     if ΔM(x_s) > th then
8:                         n(v_ij ∧ e+) = n(v_ij ∧ e+)+1
9:                     end if
10:                end if
11:            end for
12:        end for
13:    end for
14:    for all v_i, i ∈ n do
15:        for all v_ij, j ∈ m do
16:            p(e+|v_ij) = n(v_ij ∧ e+) / n(v_ij)
17:        end for
18:    end for
19: end procedure
```

The initial structure of the BN and the conditional probabilities are learnt based on the data from the k parallel optimisation instances. Figure 9 shows the steps of the process. This probabilistic effect assessment strategy calculates the relative success rate of each parameter value with respect to other parameter values. The calculated conditional probabilities deal with the uncertainties that arise from the stochastic nature of GAs by measuring the effect of parameters probabilistically, instead of using directly the performance difference.

5.1 Example of Bayesian Parameter Control

To illustrate the parameter effect assessment procedure described in Algorithm 9, we use an artificial data set with three categorical variables: two controlled parameters (v_1, v_2) with two intervals: v_{11}, v_{12}, v_{21}, v_{22} respectively, and the performance of the algorithm e, with two possible values: e^+ (successful) and e^- (unsuccessful). For the purpose of this example, we use 10 parallel algorithm instances $\{x_1, x_2, ...x_{10}\}$, as shown in Figure 10, with different parameter values for the two controlled parameters (v_1 and v_2) randomly sampled from the four intervals. The dots symbolically represent the solution sets of the instances.

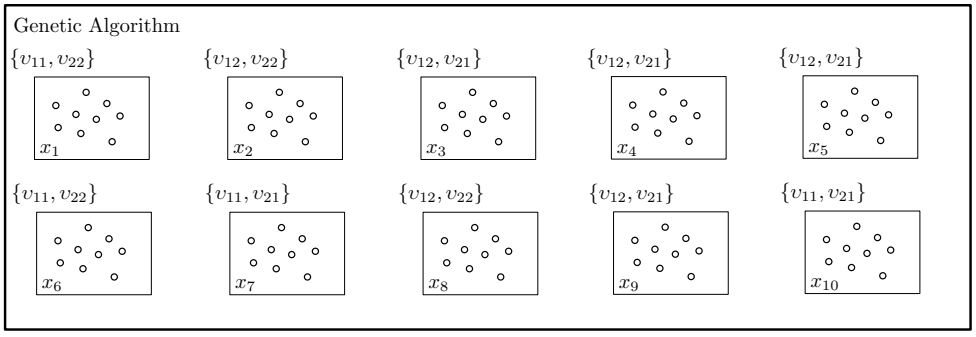


Fig. 10: Algorithm instances with parameter values selected from different intervals.

After running the GA with the different instances, the performance of each instance is assessed separately. Suppose that instances $\{x_1, x_2, x_7, x_{10}\}$ are deemed successful as shown in Figure 11. The same results are written in a table format as depicted in Table 1. Using the data in Table 1, we calculate the frequency of each parameter value in the 10 instances. Results are shown in Table 2a.

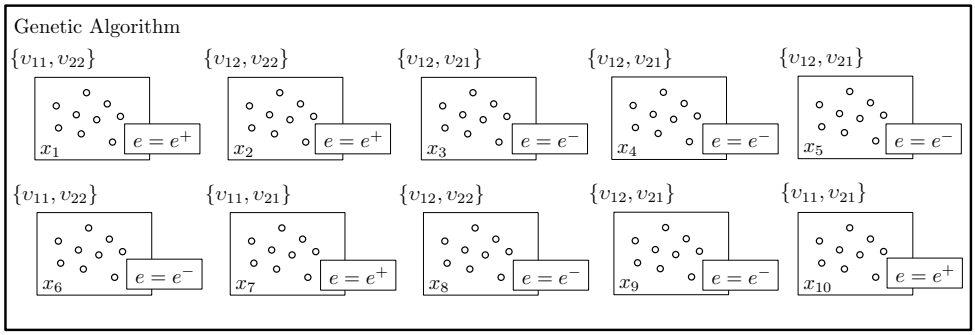


Fig. 11: Successful algorithm instances.

Table 1: Performance of algorithm instances with different values for parameters v_1 and v_2.

Instance	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
v_1	v_{11}	v_{12}	v_{12}	v_{12}	v_{12}	v_{11}	v_{11}	v_{12}	v_{12}	v_{11}
v_2	v_{22}	v_{22}	v_{21}	v_{21}	v_{21}	v_{22}	v_{21}	v_{22}	v_{21}	v_{21}
e	e^+	e^+	e^-	e^-	e^-	e^-	e^+	e^-	e^-	e^+

Table 2: Overall frequencies of parameter values, and frequencies of parameter values in the successful and unsuccessful instances.

(a) Overall frequencies.

$n(v_{11}) = 4$	
$n(v_{12}) = 6$	
$n(v_{21}) = 6$	
$n(v_{22}) = 4$	

(b) Frequencies in the successful and unsuccessful instances.

$n(v_{11} \wedge e^+) = 3$	$n(v_{11} \wedge e^-) = 1$
$n(v_{12} \wedge e^+) = 1$	$n(v_{12} \wedge e^-) = 5$
$n(v_{22} \wedge e^+) = 2$	$n(v_{21} \wedge e^-) = 4$
$n(v_{21} \wedge e^+) = 2$	$n(v_{22} \wedge e^-) = 2$

(c) The conditional probability of each parameter value.

$p(e^+\|v_{11}) = 3/4$	$p(e^-\|v_{11}) = 1/4$
$p(e^+\|v_{12}) = 1/6$	$p(e^-\|v_{12}) = 5/6$
$p(e^+\|v_{21}) = 1/3$	$p(e^-\|v_{21}) = 4/6$
$p(e^+\|v_{22}) = 1/2$	$p(e^-\|v_{22}) = 2/4$

The next step involves calculating the frequency of each parameter value in the successful and unsuccessful instances, as shown in Table 2b. The frequencies of parameter values in the successful algorithm instances indicate that parameter values v_{21} and v_{22} have a similar effect in the successful performance of the algorithm. However, it is important that we also consider the cases in which these parameter values were unsuccessful. The conditional probabilities calculated in the final step of the parameter effect assessment strategy using Equation 3 incorporate this information. Results are shown in Table 2c.

The data in Table 2c shows that although parameter values v_{22}, v_{21} have the same frequency in the successful algorithm instances, they do not have the same success rates, i.e. their effect on the successful performance of the algorithm is not the same. The conditional probabilities help in deriving conclusions about the causes of the successful or unsuccessful performance of the algorithm instances. For example, the success rate of parameter value v_{11} is $p(e^+|v_{11}) = 3/4$, which is higher than the success rate of parameter value v_{12} ($p(e^+|v_{12}) = 1/6$). This indicates that parameter value v_{11} has a higher probability of being the cause of the successful performance of the algorithm instances.

6 A Case Study of an Automotive Embedded System

The trends in automotive domain feature standardisation of the embedded software and hardware structure. In the future cars will have more centralised, multi-functional and multi-purpose hardware, with less dedicated sensors, Electronic Control Units (ecus) and actuators (Broy, 2006). The current ecus in cars often multiplex several sub-applications (Broy, 2006). Ecus, sensors, actuators, software, hardware and mechanical devices are no longer developed as one integrated piece. As a result, software has become an independent sub-product in the automotive domain (Broy, 2006).

One important initiative towards the standardisation of the automotive systems is the AUTomotive Open System ARchitecture (AUTOSAR) project, which aims at making possible the reuse of software components between different vehicle platforms (Scharnhorst et al., 2005). To achieve this the project includes the standardisation of basic system functions and functional interfaces, the ability to integrate and transfer functions and especially improve software updates and upgrades over the vehicle lifetime (Heinecke et al., 2004). AUTOSAR provides a common software infrastructure for automotive systems of

all vehicle domains based on standardized interfaces. An integrated approach for software-based vehicle systems would help not only in supporting dependable real-time execution and the management of change, but also the integration of software from different suppliers (Cuenot et al., 2007). Our approach is inspired by the AUTOSAR initiative.

A systematic literature review of 188 papers on software architecture optimisation methods was conducted prior to this case study (Aleti et al., 2013), which showed that most of the approaches use examples (27% of overall investigated papers) to demonstrate of evaluate the contributions made. Only 16% of the papers use academic case studies (16%)or industrial case studies (16%), which emphasises the requirement for more real-life case studies in this area, as also suggested by Kitchenham et al. (1995).

6.1 Design and Implementation of the Case Study

For the design and implementation of this case study, we followed the guidelines proposed by Verner et al. (2009), and cross-checked our work with the recommendations and examples from Runeson et al. (2012). The main steps followed during the construction of the case study include (i) refining of research objectives and the formalisation of research questions, (ii) defining research propositions, (iii) building a conceptual framework, defining concepts, measures, and deciding the methods of data collection, (iv) deciding how the results will be analysed, (v) describing the case study and defining it's boundaries, and (vi) identifying the appropriate level of confidence on the findings.

The Genetic Algorithm with the Bayesian parameter control strategy presented in Section 3 is employed to optimise two problem from the automotive industry: a redundancy allocation problem and a component deployment problem. The quality of the embedded architectures found by the Adaptive Genetic Algorithm is compared to the ones found by a GA without adaptation. The objectives of this case study are to (i) 'Investigate the effect of parameter values on algorithm performance when applied to an embedded system from the industry.', and (ii) 'Assess the performance of the proposed adaptive parameter control on a real-life system'. These two objectives lead to the following research questions: (i) 'Do parameter values affect the performance of a Genetic Algorithm when solving a real-life problem in Software Engineering? ', and 'Does using an adaptive parameter control strategy produce better results than pre-tuned parameter values?'.

The ensure sufficient coverage of sources of data, and enhance the reliability of the findings, the case-study was built with inputs from two different organisations: the Australian Automotive Industry via the cooperate research centre (AutoCRC), and Volvo Sweden. The process involved consultations and getting feedback on the hardware architecture, software architecture, the parameters involved and their values. The design was improved iteratively, and we found it necessary to remove, add, and change the different aspects involved. The draft case study was peer reviewed by colleagues not involved in the project, to ensure its quality.

The design of the case study involves four main stages: (i) the design of the hardware architecture, its properties and the real-life values, (ii) the design of the software architecture, which also includes the properties of the software elements and their values, (iii) the formulation of the redundancy allocation problem, which includes the quality attributes that require optimisation, and constraints, and (iv) the formulation of the component deployment problem, which requires different quality attributes and constraints. These four stages are described in detail in the following sections. To check for a statistical significance of the findings, a set of experiments were conducted, where the problem instances were generated using the properties identified in the case study.

6.2 Hardware Architecture

In the automotive industry, an existing hardware topology is usually used, because car models remain the same through parts of their lifetimes. The hardware is the platform where software components run, which forms the basis of the increasingly sophisticated functionality of contemporary cars. The hardware model of an automotive embedded system is composed of a distributed set of Electronic Control Units (ECUs), which have different capacities of memory, processing power, access to sensors, etc. Automotive embedded systems closely interact with the physical environment, typically via sensors and actuators. All the hardware modules (ECUs, sensors, actuators) are connected through communication buses, which are shared among the hardware units, such as Ethernet, CAN, or FlexRay. The hardware modules and the communication buses form the hardware architecture of an automotive system. An example is depicted in Figure 12.

Many types of buses can be present, having different characteristics of data rates and reliability. For example, for the Brake-by-wire (BBW) system a highly reliable bus is used due to the safety-critical nature of this system, whereas a less reliable bus may be sufficient for multi-media streaming. Each ECU has access to different sensors and actuators, which impose localisation constraints for the software components. In other words, software components that read from a specific sensor or write to a particular actuator, have to be deployed to an ECU that has access to the sensor or actuator.

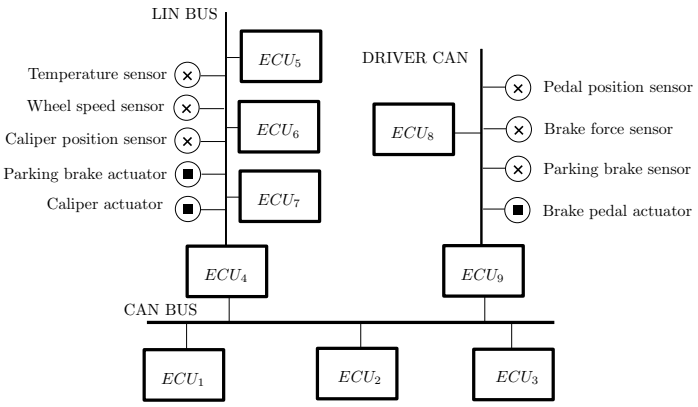


Fig. 12: The hardware architecture of an automotive system.

The hardware elements are annotated with different properties required for the calculation of the quality attributes. The communication among the hardware hosts is achieved by the network links, which have different characteristics of data rates, reliability, etc. Figure 12 depicts three different network links: a LIN bus, a Driver CAN and a CAN bus. A description of the properties of hardware units and network links are depicted in Table 3.

Table 3: Properties of the hardware elements.

Annotation	**Definition**	**Description**	**Range**
$ps(h_i)$	$\mathcal{H} \rightarrow \mathbb{N}$	Processing speed of the host in $MIPS$	$[2, 16]$
$mh(h_i)$	$\mathcal{H} \rightarrow \mathbb{N}$	Memory capacity of host h_i	$[2^6, 2^{10}]$
λ_h	$\mathcal{H} \rightarrow \mathbb{R}$	Failure rate of the host	$[e^{-2}, e^{-6}]$
λ_n	$\mathcal{N} \rightarrow \mathbb{R}$	Failure rate of the network link	$[e^{-2}, e^{-5}]$
bw	$\mathcal{N} \rightarrow \mathbb{N}$	the bandwidth of a network link	$[2^6, 2^10]$
tt	$\mathcal{N} \rightarrow \mathbb{N}$	the transfer time for a link per execution in ms	$[1, 100]$

The properties of the hardware elements, described in Tables 3 are configured using values from a real world applications of an automotive embedded system. The configuration of the hardware and software architecture are given in Tables 4-6.

Table 4: Values of the hardware architecture for the BBW system

(a) Values of hardware nodes.

Host	**mh**	**ps**	λ_h	**lr**
ecu_1	512	4	4.0e-4	0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1
ecu_2	1024	6	4.0e-4	0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1
ecu_3	512	2	2.0e-5	0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1
ecu_4	1024	2	1.0e-4	1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0
ecu_5	510	11	8.0e-4	1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0
ecu_6	1024	11	2.0e-4	1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0
ecu_7	1024	8	6.0e-5	1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0
ecu_8	1024	7	4.0e-5	0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1
ecu_9	1024	8	8.0e-4	0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1

(b) Values of communication links.

Link	λ_n	**bw**	**tt**	**r**
Lin bus	3.0 e-05	128	10	0.9
Driver CAN	1.2 e-04	64	2	0.8
CAN bus	4.0 e-05	64	4	0.8

6.3 Software Architecture

The software layer of an automotive embedded system consists of a high number of lightweight components, representing the logical blocks of system functionality (typically in a low-level programming language). This forms the software architecture of the system. Automotive software has very diverse functionality, ranging from entertainment software to safety-critical, real-time control software. We illustrate the software architecture of an automotive system using the Brake-by-wire (BBW) service, which is a real-time system that performs safety-critical tasks.

BBW technology is a recent innovation in the automotive industry, which replaces the traditional mechanical and hydraulic control systems with electronic control systems. A diagram of the BBW software model is shown in Figure 13. Each box represents a software component and the connections among them correspond to their interactions. The direction of the arrows represents the sequence of the execution of the components. A BBW pedal is usually equipped with several sensors which provide information about the driver's brake request. The *Pedal Position Sensor* (PPS) and the *Brake Force Sensor* (BFS) measure the force applied by the driver to the brakes and the current position of the brakes. Upon measurement of the driver's brake request, the brake demand is sent to the *Brake Pedal Control* (BPC) via the communication network. The BPC transfers the brake request to the *Central Brake Control* (CBC), which generates four independent brake commands and sends them to the *Calliper Control Unit* (CCU) in each wheel. These commands are usually in the form of four clamp forces that are generated between each of the four brake discs and their corresponding brake pads. Each CCU includes a *Calliper Clamp Actuator* (CCA) to clamp the brake pad toward the braking disc and a *Parking Brake Actuator*. The CCU also processes additional inputs from several sensors, such as the *Temperature Sensor* (TS) and the *Calliper Position Sensor*, which regulate the brake command execution.

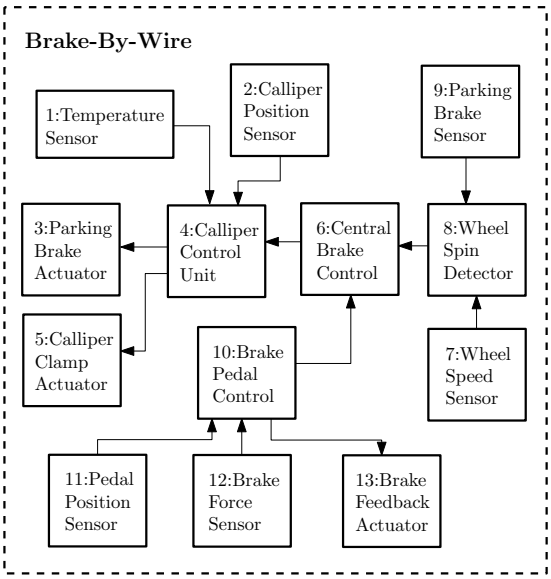


Fig. 13: The software architecture of the Brake-by-wire system.

Table 5: Properties of the software elements.

Annotation	Definition	Description	Range
$ext(c_i)$	$\mathcal{C} \to \mathbb{N}$	Estimated time taken for a single execution of component c_i	$[0.1, 1.0]$
$q_0(c_i)$	$\mathcal{C} \to \mathbb{R}$	Mean execution initialisation probability for component c_i	$[0, 1]$
$mc(c_i)$	$\mathcal{C} \to \mathbb{N}$	Memory size required to run component c_i.	
$cost(c_i)$	$C \to \mathbb{N}$	The price of component c_i in \$	$[20, 50]$
$p(c_j, c_i)$	$\mathcal{C} \times \mathcal{C} \to \mathbb{R}$	Transition probability from component c_j to component c_i	$[0, 1]$
$ms(c_j, c_i)$	$\mathcal{C} \times \mathcal{C} \to \mathbb{N}$	Message size transferred from component c_j to component c_i	$[2, 2^5]$
$cf(c_i, c_j)$	$\mathcal{C} \times \mathcal{C} \to \mathbb{R}$	Communication frequency between components c_i and c_j	$[0, 1]$
$lr(c_i, h_j)$	$\mathcal{C} \times \mathcal{H} \to \{0, 1\}$	Localisation restrictions. The restriction $lr(c_i, h_j) = 1$ if component c_i has to be deployed to hardware host h_j, otherwise $lr(c_i, h_j)$ is equal to 0	{0, 1}
$cr(c_i, c_j)$	$\mathcal{C} \times \mathcal{C} \to \{0, 1\}$	Collocation restrictions. Restriction $cr = 1$ if c_i has to be deployed in the same hardware unit as c_j, and $cr = 0$ if there is no such restriction	{0, 1}

There are also different sensors and a controller to tune the actual brake force to the desired clamp force received from the *Central Brake Control* (CBC), such as the *Parking Brake Sensor* (PBS) and the *Wheel Speed Sensor* (WSS). After the brake request has been applied, a message is generated by the *Brake Feedback Actuator* (BFA) which notifies the driver. A description of the properties of software elements is given in Table 5. The values of the case-study for the software elements are presented in Table 5. The values of the interactions between components and the collocation restrictions are shown in Table 6b and 6c. Collocation restrictions dictate which components should be deployed together in the same hardware host. In the current case-study, component c_8 has to be deployed in the same hardware host as component c_7, and components c_{11} and c_{12} have to be deployed together in the same hardware host. Usually, in automotive system, collocation restrictions are not very hard, i.e. not many software components are required to be collocated.

The software architecture is deployed to the hardware architecture described in Section 6.2, to realise the BBW system. This process involves two main design steps which are optimised individually, described in the following sections. First a redundancy level is decided for each component. Then the software

Table 6: Values of the software architecture for BBW system.

(a) Software components.

	Component	**mc**	q_0	**ext**	**cost**
c_1	Temperature Sensor (TS)	64	0.108	12	25
c_2	Calliper Position Sensor (CPS)	128	0.182	10	30
c_3	Parking Brake Actuator (PBA)	64	0	2	20
c_4	Calliper Control Unit (CCU)	512	0	20	30
c_5	Calliper Clamp Actuator (CCA)	256	0	5	40
c_6	Central Brake Control (CBC)	1024	0	10	30
c_7	Wheel Speed Sensor (WSS)	64	0.1	12	35
c_8	Wheel Spin Detector (WSD)	128	0.2	8	20
c_9	Parking Brake Sensor (PBS)	128	0.1	4	30
c_{10}	Brake Pedal Control (BPC)	512	0	6	35
c_{11}	Pedal Position Sensor (PPS)	64	0.13	10	30
c_{12}	Brake Force Sensor (BFS)	128	0.18	10	35
c_{13}	Brake Feedback Actuator (BFA)	64	0	4	25

(b) Interactions.

c_i	c_0	c_1	c_3	c_3	c_5	c_7	c_6	c_8	c_9	c_{10}	c_{11}	c_9
c_j	c_3	c_3	c_2	c_4	c_3	c_5	c_7	c_7	c_5	c_9	c_9	c_{12}
p	1	1	0.3	0.7	1	1	1	1	0.6	1	1	0.4
cl	2	2	2	2	2	1	2	1	2	1	2	2
ext	0.2	0.3	0.1	0.4	0.3	0.2	0.6	0.4	0.2	0.3	0.5	0.4
es	10	12	12	10	12	6	12	6	12	8	10	6
cf	1	1	0.3	0.7	1	1	1	1	0.6	1	1	0 .4

(c) Collocation restrictions between components.

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}
c_1	0	0	0	0	0	0	0	0	0	0	0	0	0
c_2	0	0	0	0	0	0	0	0	0	0	0	0	0
c_3	0	0	0	0	0	0	0	0	0	0	0	0	0
c_4	0	0	0	0	0	0	0	0	0	0	0	0	0
c_5	0	0	0	0	0	0	0	0	0	0	0	0	0
c_6	0	0	0	0	0	0	0	0	0	0	0	0	0
c_7	0	0	0	0	0	0	0	1	0	0	0	0	0
c_8	0	0	0	0	0	0	1	0	0	0	0	0	0
c_9	0	0	0	0	0	0	0	0	0	0	0	0	0
c_{10}	0	0	0	0	0	0	0	0	0	0	0	0	0
c_{11}	0	0	0	0	0	0	0	0	0	0	0	1	0
c_{12}	0	0	0	0	0	0	0	0	0	0	1	0	0
c_{13}	0	0	0	0	0	0	0	0	0	0	0	0	0

components and their redundancies are allocated to the ecus. In each step, different quality attributes are optimised.

6.4 Redundancy Allocation Optimisation

Allocating redundancies to software components adds communication and processing overhead, as well as requirements for additional hardware (e.g. sensors). This problem has been addressed mostly in component based software architecture development. In this work we focus on three quality attributes: the added cost, the improvement in the reliability of the systems, and the overheads in the response time. To quantify the quality of a single redundancy allocation architecture ra, we define three objective functions $F : RA \rightarrow \mathbb{R}^2$, where $F(ra) = (rt(ra), r(ra), cost(ra))$ s.t. $rt(ra)$ is the the response time of ra, defined by Equation 4, $r(ra)$ denotes the reliability (probability of failure-free operation) of ra, defined in Equation 6 and $cost(ra)$ is the total cost of the system with the redundancy levels assigned to components.

Response time: The response time has already been the focus of an extensive research in the embedded systems (Sharma et al., 2005; Fredriksson et al., 2007; Sharma and Trivedi, 2007). To estimate the response time of a software architecture, we employ Discrete Time Markov Chain model (Sharma and Trivedi, 2007). The system behaviour is described as a DTMC with probabilities of execution transfer between

components together with probabilities of execution initialisation at each component. The response time rt for a redundancy allocation candidate ra is calculated as follows:

$$rt(ra) \ = \ \sum_{i \in n} ext(c_i) \cdot exv(c_i) \tag{4}$$

where $ext : C \rightarrow \mathbb{N}$ is the estimated time taken for a single execution of a component, and $exv : C \rightarrow \mathbb{R}$ quantifies the expected number of executions of a component during the system execution. This can be computed by solving the following equation (Kubat, 1989):

$$exv(c_i) = q_0(c_i) + \sum_{j \in \setminus} (exv(c_j) \cdot p(c_j, c_i)) \tag{5}$$

where $p(c_j, c_i)$ denotes the transition probability from component c_j to component c_i, and $q_0(c_i)$ represents the mean execution initialisation probability for component c_i.

Reliability: We employ a well-established method of reliability estimation presented by Kubat (Kubat, 1989; Goševa-Popstojanova and Trivedi, 2001). In estimating the reliability of a single component, we assume that failure of a component has an exponential distribution (Shatz et al., 1992), which is characterised by the failure rate parameter λ. First, we calculate the *reliability of a single component* c_i as defined by Shatz et al. (Shatz et al., 1992), computed as $r(c_i) = e^{-\lambda(c_i) \cdot ext(c_i)}$. The reliability of a component with its replicas connected in parallel for the architectural alternative ra is equal to $r(c_{i,rep}) = 1 - (1 - r(c_i))^{ra(c_i)+1}$. The reliability of transferring a message from component c_i to component c_j is calculated as $r(c_i, c_j) = e^{-\lambda(c_i, c_j) \cdot tt(c_i, c_j)}$, where $tt(c_i, c_j)$ is the transfer time for a link per execution and $\lambda(c_i, c_j)$ is the failure rate in the communication. The presence of multiple senders increases the reliability (due to the tolerance against commission and value failures), which is calculated as $r(c_{i,rep}, c_j) = 1 - (1 - r(c_i, c_j))^{ra(c_i)+1}$.

The reliabilities of individual system elements (subsystems and links) for a single visit are used to compute the reliability of the system execution based on the expected number of executions (Goševa-Popstojanova and Trivedi, 2001; Kubat, 1989):

$$r(ra) \ \approx \ \prod_{i \in n} (r(c_{i,rep}))^{exv(c_i)} \tag{6}$$

Cost: The cost of the system for each architecture alternative is evaluated as the sum of the costs of individual components and the respective redundancies as follows:

$$cost(ra) \ = \ \sum_{i \in n} cost(c_i) \cdot (ra(c_i) + 1) \tag{7}$$

6.5 Component Deployment Optimisation

Another challenge for the automotive industry remains the deployment function which relates hardware to software. The hardware, software and the deployment function should be a concrete realisation of the logical architecture, which describes the interaction between the logical components. The software components run on ecus and communicate by bus systems. The deployment process determines the ECU to which each software component is allocated.

Generally, the same platform can be used for different models and different products at the same time, by integrating new functionality in an iterative process, which makes the implementation of new innovative functions the main focus of the automotive electronics (Scharnhorst et al., 2005). To understand how the deployment architecture affects the quality of the final embedded systems in a car, consider the two deployment architectures shown in Figure 14.

In the first deployment architecture, frequently interacting software components have been deployed into the same hardware resource, which result in an architecture with a lower communication overhead compared to the second deployment architecture. However, the first deployment architecture produces a longer scheduling length due to the sequential execution of the components that require information from each other. To quantify the quality of a single deployment architecture d, we define three objective

functions $F : D \rightarrow \mathbb{R}^3$, where $F(d) = (sl(d), co(d), dtr(d)$ s.t. $sl(d)$ is the quantification of the scheduling length of d, defined by Equation 8, $co(d)$ denotes the communication overhead of d, defined in Equation 9 and $dtr(d)$ denotes the data transmission reliability defined in Equation 10.

Scheduling length: The electronic control units (ecus) are assumed to have a fixed and deterministic scheduling, which is a technology used with Time Triggered Architectures in embedded systems in order to maintain the predictability of the internal behaviour of the system (Heiner and Thurner, 1998). With this scheduling strategy, when the software components are allocated to ecus, a fixed schedule is determined. Each component is given a specific time frame, which it requires to complete the execution. The schedule is done in a round-robin fashion, i.e. each component is allocated its own time frame in a circular fashion. The sum of all the execution time slots in the ECU compose the *scheduling length*, calculated as:

$$sl(d) = \frac{1}{m} \cdot \sum_{j=1}^{m} \left(\frac{\sum_{c \in C_{h_j}} ext(c)}{ps(h_j)} \right). \quad (8)$$

where C_{h_j} is the set of components deployed into the ECU h_j, $ext(c)$ is the estimated time taken for a single execution of the component c and $ps(h_j)$ is the processing speed of the ECU h_j.

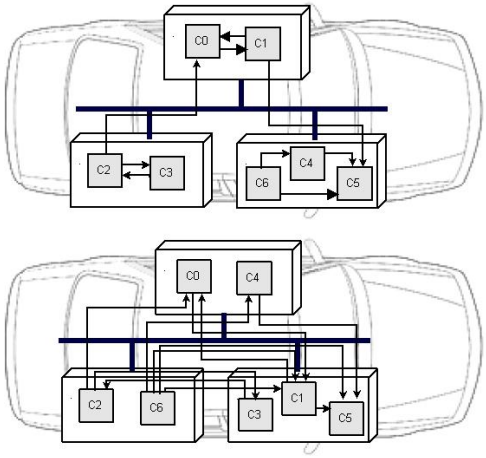


Fig. 14: Two potential deployment architectures resulting in different quality attributes.

Communication overhead: In embedded systems with constrained hardware resources, repeated transmissions between software components are discouraged. The Communication Overhead (CO) objective attempts to enforce minimal data communication for a given set of components and system parameters. As a network- and deployment-dependent metric, the overall communication overhead of the system is used to quantify this aspect. This metric was first formalised by Medvidovic and Malek (Medvidovic and Malek, 2007), defined as:

$$co(d) = \sum_{i=1}^{n} \sum_{j=1}^{n} cf(c_i, c_j) \cdot nd(d(c_i), d(c_j)) + \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{cf(c_i, c_j) \cdot ms(c_i, c_j)}{bw(d(c_i), d(c_j)) \cdot r(d(c_i), d(c_j))} \quad (9)$$

where $ms : C \times C \rightarrow \mathbb{N}$ is the component messages size, with $ms(c_i, c_j) = 0$ if $c_i = c_j$ or if there is no communication between the two components, $cf : C \times C \rightarrow \mathbb{R}$ is the communication frequency between c_i and c_j, $bw : H \times H \rightarrow \mathbb{N}$ is the network bandwidth, with $bw(h_i, h_j) = 0$ if $h_i = h_j$ or there is no network connection between h_i and h_j, and $nd : H \times H \rightarrow \mathbb{N}$ is the network delay, with $nd(h_i, h_j) = 0$ if $h_i = h_j$ or there is no network connection between h_i and h_j.

Data transmission reliability: In a deployment architecture, the communication of components that compose the same service and that are deployed into different ecus is supported by the network. If components have to exchange frequent messages with each other, the reliability of the data transmission in the network becomes a crucial quality attribute. Data transmission reliability is especially important in a real-time embedded system, since important decisions are taken based on the data transmitted through the communication links. Following the definition introduced by Malek (Malek, 2007), data transmission reliability $(dtr(d))$is calculated as follows:

$$dtr(d) = \sum_{i=1}^{n} \sum_{j=1}^{n} cf(c_i, c_j) \cdot r(d(c_i), d(c_j)) \quad (10)$$

where $cf(c_i, c_j)$ is the communication frequency between components c_i and c_j, and $r(d(c_i), d(c_j))$ is the reliability of the communication link between the hardware resources where c_i and c_j are deployed, which is equal to $e^{-\lambda_n(d(c_i),d(c_j)) \cdot tt(d(c_i),d(c_j))}$, where $tt(d(c_i), d(c_j)$ is the transfer time for a link per execution and $\lambda_n(d(c_i), d(c_j))$ is the failure rate in the communication network.

6.6 Constraints

Not all deployment architectures represent feasible alternatives. For instance, placing all components into a single host is not feasible due to memory constraints. Hence the optimisation procedure should check if every new created solution satisfies the set of constraints. The set of constraints Ω is defined independently of the quality functions. In this work, we consider three constraints $\Omega = \{mem, loc, colloc\}$, where mem is the memory constraint, loc denotes the localisation constraint and $colloc$ is the collocation constraint.

Memory constraint: Processing units have limited memory, which enforces a constraint on the possible components that can be deployed in each ecu. Formally, let $d^{-1} : \mathcal{H} \rightarrow \mathcal{C}_h$ denote the inverse relation to $d \in D$, i.e. $d^{-1}(\mathcal{H}) = \{C_h \in C \mid d(C_h) = h\}$. Then the memory constraint $mem : D \rightarrow \{true, false\}$ is defined as follows:

$$mem(d) \;=\; \forall h \in \mathcal{H} : \sum_{C_h \in d^{-1}(h)} mc(C_h) \;\leq\; mh(h) \tag{11}$$

where $mc(C_h)$ is the total memory required to run the set of components deployed to the hardware host h, and $mh(h)$ is the available memory in host h. In other words, this constraint does not allow any deployment solution which exceeds the available memory in the hardware resources.

Localisation constraints: Processing units have access to different sensors and actuators which are used by the software components. For instance, the Brake Force Sensor (BFS) software component reads information from the respective sensor. As a result it has to be deployed into an ECU which can communicate with that sensor (i.e. is connected via the network or the sensor is built into the ECU). The availability of a specific sensor in a hardware host, restricts the allocation of the software component that uses the sensor to that particular host. This constraint is called localisation constraint, denoted $loc : D \rightarrow \{true, false\}$ and is defined as follows:

$$loc(d) = \forall c \in C : (h \in lr(c) \;\Rightarrow\; d(c) \neq h) \tag{12}$$

where $lr(c)$ is the list of localisation restrictions.

Collocation constraints: Some of the components should not be allocated in the same ecu. For example, a software component should not be allocated into the same ECU as its redundancies, so that if one of the ecus fails the other component can still perform the required task. This is called collocation constraint, denoted as $colloc : D \rightarrow \{true, false\}$, and restricts the allocation of two software components to two different hosts. Collocation constraint is calculated as follows:

$$colloc(d) = \forall c \in C : (h \in cr(c_i, c_j) \;\Rightarrow\; d(c_i) \neq d(c_j) \tag{13}$$

where $cr(c_i, c_j)$ is the matrix of collocation restrictions.

6.7 Results of the Optimisation Process

The two software design optimisation problems, the optimisation algorithms used to solve them and the parameter control methods are implemented in ArcheOpterix (Aleti et al., 2009a), which is a generic platform for modelling, evaluating and optimising embedded systems. The presented case-study is a comparatively small segment of the actual automotive architecture optimisation problem. Despite this fact, the possible number of candidate architectures is still too large to search with an exact algorithm, i.e. $9^{13} \approx 2.54 \cdot 10^{12}$ options for the redundancy allocation problem, and even larger for the component deployment problem. To optimise both architecture optimisation problems we employed, Non-dominated Sorting Genetic Algorithm-II (NSGA-II).

NSGA-II is a Genetic Algorithm which has been demonstrated as one of the most efficient algorithms for multi-objective optimization on a number of benchmark problems (Deb et al., 2002). A brief description of the basic components of NSGA-II will be given in this section. The NSGA-II algorithm and its detailed implementation procedure is described by Deb et al. (2002). The algorithm uses non-dominated sorting for fitness assignments. All individuals that are not dominated by any other individuals, are assigned to front number 1. All individuals only dominated by individuals in front number 1 are assigned front number 2, and so on. A mating pool is then created with solutions selected from the population according

to the ranks that has been assigned to them during the ranking process. The solutions with a lower rank have a higher chance of being selected to be part of the mating pool than the ones with a higher rank value. This helps the quick convergence of the algorithm towards the optimal solutions. The mating pool will then serve for the random selection of the individuals to reproduce using the genetic operators,

The Bayesian parameter control strategy is used to adjust the parameter values of NSGA-II. To demonstrate the usefulness of the proposed parameter control method, the performance of NSGA-II with parameter control was compared to the performance of NSGA-II with pre-tuned parameter values. Mutation rate, crossover rate, mating pool size, crossover operator, and mutation operator used for the benefit of these experiments, are probably the most conspicuous control parameters to optimise in stochastic optimisation (Eiben and Schut, 2008). The population size was not considered in the current work, and a pre-tuned value of 100 was employed, which was selected after exploring values in the range $[50, 500]$. Based on preliminary exploration, the adopted ranges/values of the parameters controlled in this experiment are given in Table 7a.

Table 7: The tuned and controlled ranges/values of algorithm parameters.

(a) Controlled parameter ranges/values.

Parameter	**Range/value**
Mutation rate	[0.0010,0.5]
Crossover rate	[0.5,0.9]
Mating pool size	[0.1,0.7]
Mutation operator	Single-point, Uniform
Crossover operator	Single-point, Uniform

(b) Tuned parameter values for Component Deployment and Redundancy Allocation Problems.

Parameter	**CDP**	**RAP**
Mutation rate	0.1	0.2
Crossover rate	0.58	0.8
Population size	100	100
Mating pool size	0.6	0.6
Mutation operator	Single-point	Uniform
Crossover operator	Single-point	Uniform

The tuning of the static parameter values was performed following recommendations of Smit and Eiben (Smit and Eiben, 2009). We use a Sequential Parameter Optimisation (SPO) (Bartz-Beielstein et al., 2005), which tests each parameter combination using several runs. To decrease the number of tests required, we employ a racing technique, which uses a variable number of runs depending on the performance of the parameter configuration. Parameter configurations are tested against the best configuration so far, using at least the same number of function evaluations as employed for the best configuration. Five different design points are selected for mutation rate, crossover rate and mating pool size from the ranges depicted in Figure 7a. The results from the tuning process are shown in Table 7b.

For the purpose of these experiments, each instance for the Bayesian parameter control has 10 solutions, which means that BGA uses 10 instances for each run (i.e. 10 instances $\times$ 10 solutions = 100 solutions). The execution of the algorithm was set to 10 000 candidate evaluations, and performed on a dual-core 2.26 GHz processor computer. The algorithm took 92 seconds for the 10 000 function evaluations and generated 231 approximate solutions, four of which are depicted in Table 8.

Table 8: Redundancy allocation solutions.

Sol	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	$c(ra)$	$rt(ra)$	$r(ra)$
ra_1	2	1	2	2	1	1	1	2	1	1	1	1	2	890	56.48	0.996099
ra_2	2	2	2	2	2	2	2	2	2	1	2	2	2	1120	62.96	0.996347
ra_3	0	0	0	0	0	0	0	0	1	0	0	0	1	440	41.15	0.793488
ra_4	0	0	1	1	0	0	0	0	0	0	0	0	0	435	43.27	0.920808

As it can be seen from the results in Table 8, the second solution (ra_2) has the highest reliability among the four. However, its cost and response time are also the highest. The first solution (ra_1) is lower in cost and has a smaller response time compared to the second solution, however its reliability is also lower. The two last solutions (ra_3 and ra_4) have the lower cost and response time than the first two solutions, but their reliability values are much lower.

Both solutions ra_3 and ra_4 have only two components with one redundancy level assigned (solution ra_3 has components c_9 and c_{13}, whereas solutions ra_4 has components c_3 and c_4). Interestingly, the reliability of the ra_4 is much higher than the respective value of ra_3 ($0.920808 > 0.793488$). The response

time and the cost of solution ra_4 are also higher than ra_3, although the difference between respective values is not big.

Table 9: Component deployment solutions.

Sol	c_1	c_2	c_{31}	c_{32}	c_{41}	$c_{4,1}$	c_5	c_6	c_7	c_8	c_{12}	c_{13}	c_9	c_{10}	c_{11}	$sl(d)$	$co(d)$	$dtr(d)$
d_1	6	6	3	6	3	3	3	3	7	7	9	8	8	8	8	4.21	65.28	0.955
d_2	6	6	6	6	2	2	2	2	6	6	9	9	9	9	9	1.96	74.81	0.951
d_3	6	6	6	6	2	2	6	2	6	6	9	9	9	9	9	1.91	94.68	0.941
d_4	6	6	2	6	2	2	6	2	6	6	9	9	9	9	9	1.93	82.81	0.945

At this stage, the system designer would have to select the architecture representing the redundancy levels for all components according to her/his preference. Although all solutions are non-dominated, i.e. none of them is worse than the others in all objective and each of them is the best in at least one objective, the system designer may usually care about some objectives more than the others. For instance, if reliability is a crucial quality attribute, (s)he may think that paying more for a more reliable architecture is agreeable.

For the purpose of this case-study, we select solution ra_4 as the final software architecture with the redundancy levels. Next, the software components have to be deployed into the hardware architecture. Similar to the redundancy allocation optimisation problem, we employ the Genetic Algorithm with the Bayesian parameter control method (BGA) to optimise the deployment of the software architecture (solution ra_4) to the hardware architecture (c.f. Figure 12). The GA reported eight nondominated deployment architectures after the optimisation process. We show four of these deployment architectures in Table 9. The optimised deployment architectures make a trade-off among the quality attributes of response time, communication overhead and data transmission reliability. The system architect has to select the architecture, which he/she thinks suitable.

Table 10: Means, variances and Kolmogorov-Smirnov test values of hypervolume indicators for the 50 runs of each problem instance using different optimisation schemes.

	Mean		**Variance**		**KS test**		**Effect**
Problem	**GA**	**BGA**	**GA**	**BGA**	**d-value**	**p-value**	**size**
Redundancy allocation	0.2617	**0.2619**	9.770E-05	1.173E-04	0.3284	0.049	0.68
Component deployment	0.2618	**0.2620**	1.179E-04	1.083E-04	0.3667	0.026	0.66

To understand the benefits of using the Genetic Algorithm with the Bayesian parameter control method (BGA) for the architecture optimisation problem, we compared the outcomes of a GA using parameter control with the same GA implementation that uses pre-tuned parameter values. Approximate algorithms are not expected to deliver exact and repeatable results, but to provide good approximate solutions where exact approaches cannot be devised. Hence, results concerning the performance of approximate algorithms are usually reported as mean values over repeated trials. We performed 50 runs for each optimisation scheme.

Different metrics for measuring the performance of multiobjective optimisation methods exist. Zitzler et al. (2003) provide a comprehensive review of quality indicators for multiobjective optimisation, finding that many commonly used metrics do not reliably reflect the performance of an optimisation algorithm. One of the few recommended metrics is the hypervolume indicator, which measures the hypervolume between the approximate solutions and a fixed reference point in the result space. In the case of maximisation problems, the most intuitive reference point is zero in each dimension, which is the value used for our results. For a detailed description of hypervolume indicator see Zitzler et al. (2007). The mean and standard deviation of the 50 hypervolume values for each optimisation scheme are reported in Table 13. The GA using the adaptive parameter control consistently outperforms the GA with tuned parameter values.

The different optimisation methods are validated using the Kolmogorov-Smirnov (KS) non-parametric test (Pettitt and Stephens, 1977), which checks for a statistical significance in the difference between the performances of the algorithms. The 50 values of the hypervolume indicators were submitted to the KS analysis which resulted in a confirmation of the out-performance of the adaptive GA with a minimum

d-value of 0.3284 at a 95% confidence level. Hence, we conclude that the superior performance of the proposed Bayesian Parameter Control is statistically significant.

To quantify the difference in the performance of the two optimisation schemes, we also calculate the effect size, which is the difference between means divided by standard deviation. For both problems, the effect size is in the range medium to large, which means that the difference in the performances of the two algorithms is meaningful.

7 Experimental Evaluation

To investigate the efficiency of the Genetic Algorithm with the Bayesian parameter control method (BGA), we designed a set of experiments which compares its performance to methods that use pre-tuned parameter values over a set of problem instances. We also compared the performance of the BGA against the performance of three state-of-the-art adaptive parameter control methods: Probability Matching (PM), Dynamic Multi-Armed Bandits (DMAB), and Adaptive Pursuit (AP). These methods are described in Section 4.

7.1 Experimental Settings

The problem instances were randomly generated over realistic value-ranges from component deployment and redundancy allocation problems with different sizes and characteristics. The configurations of the 8 problem instances for the redundancy allocation problem are as follows: 15 components (c15) 33 components (c33), 67 components (c67), 89 components (c89), 123 components 123, 150 components (c150), 185 components (c185), 220 components (c220). The redundancy levels were selected between [0,4]. For the component deployment problem the following instances were created: 15 hosts-23 components (h15c23), 15 hosts-32 components (h15c32), 33 hosts-51 components (h33c51), 33 hosts-67 components (h33c67), 55 hosts-76 components (h55c76), 55 hosts-120 components (h55c120), 72 hosts-130 components (h72c130), and 72 hosts-180 components (h72c180). Unlike the case study presented in Section 6, for the purpose of these experiments the instances for the redundancy allocation problem and the component deployment problem were generated independently.

NSGA-II (Deb et al., 2002) was used as an optimisation algorithm. The parameter control methods were employed to adjust the configuration of NSGA-II during the optimisation process. To measure the performance of the multiobjective optimisation methods we employed the hypervolume indicator (Zitzler et al., 2003), described in the previous section. All algorithm instances were allowed to run for 50 000 function evaluations. Each optimisation scheme was repeated 50 times for each problem instance. Results from the 50 runs were analysed and compared using the Kolmogorov-Smirnov (KS) non-parametric test (Pettitt and Stephens, 1977). Furthermore, the effect size was reported for each experiment.

All optimisation schemes are initialised with the same 100 solutions, and generate the same number of offspring each iteration. The main difference between the optimisation schemes is the parameter values used to generate the offspring. In the case of the GA without parameter control, the same pre-tuned parameter values were used during the optimisation run. The GAs with parameter control have their parameter values adapted during the optimisation process. In the case of the Bayesian Genetic Algorithm, the 100 solutions were divided into 10 instances, each composed of 10 solutions.

7.2 Hyper-Parameters

All adaptive algorithms involve hyper-parameters, which need to be tuned depending on the optimisation problem at hand. This defines another optimisation problem, which can become quite computationally expensive if we attempt an exhaustive exploration of the search space. However, the number of hyper-parameters is usually lower than the number of algorithm parameters that should be controlled.

The tuning of the hyper-parameter values was performed using a Sequential Parameter Optimisation (SPO) (Bartz-Beielstein et al., 2005), which tests each hyper-parameter combination using several runs. To decrease the number of tests required, we employ a racing technique, which uses a variable number of runs depending on the performance of the hyper-parameter configuration. Hyper-parameter configurations are tested against the best configuration so far, using at least the same number of function evaluations as employed for the best configuration. The results from the tuning process are shown in Table 11.

Table 11: Hyper-parameters of the three adaptive methods: Dynamic Multi-Armed Bandit (DMAB), Adaptive Pursuit (AP) and Probability Matching (PM).

Benchmark	**Hyperparameter**	**Value**	**Description**
DMAB	ς	0.5	scaling factor
DMAB	γ	100	PH threshold
AP,PM	α	0.8	adaptation rate
AP,PM	β	0.8	adaptation rate
AP,PM	p_{min}	0.1	minimum selection probability

7.3 Analysis of Results

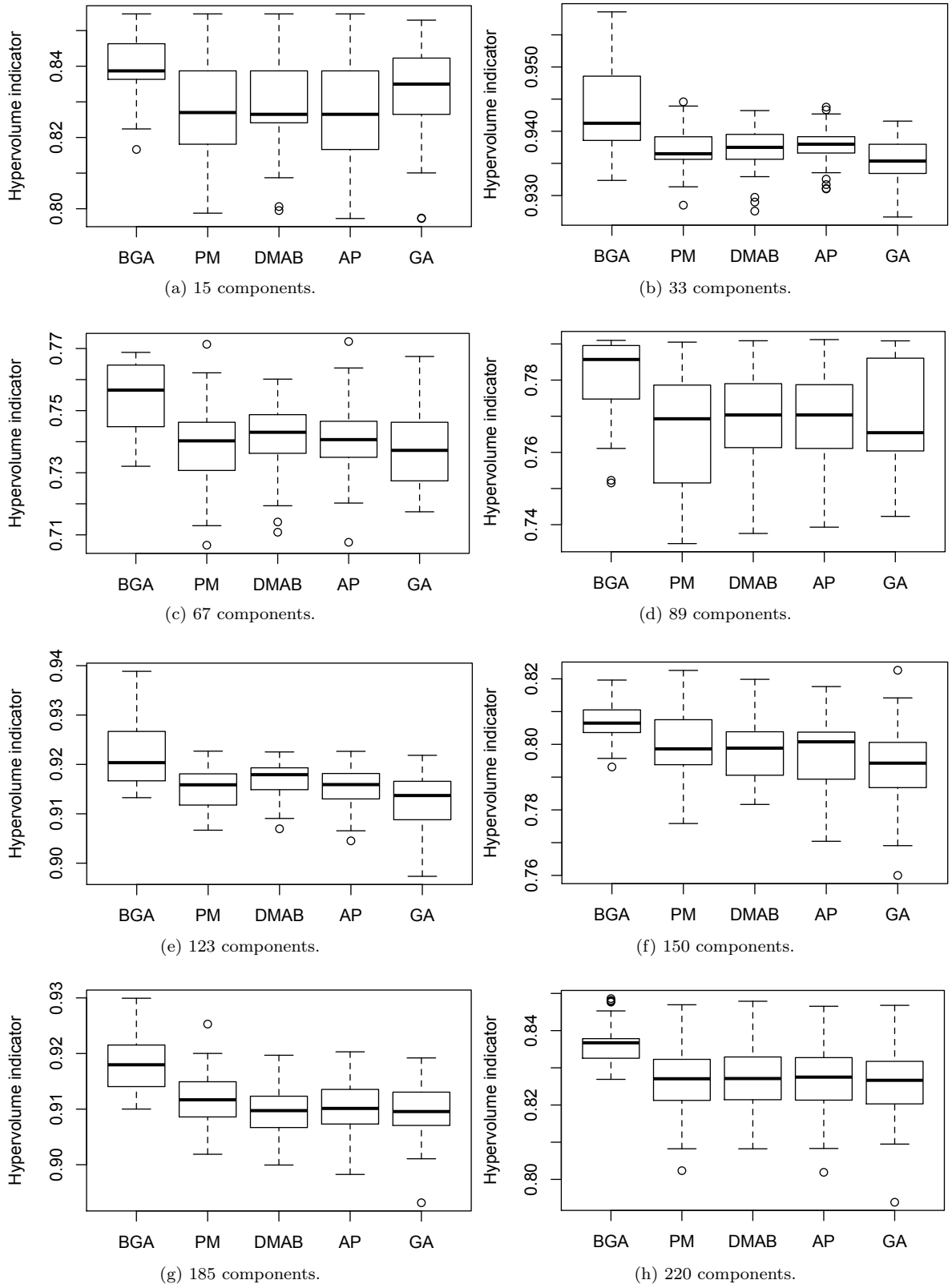
The experiments were performed on a 64-core 2.26 GHz processor computer. There was little difference in the run-times of the different optimisation schemes for the same problem instances. The main difference in run-time was observed between the different problem instances. Solving the smaller instances was obviously faster, since the evaluation of the quality attributes takes less time. The overhead from the parameter control methods is negligible compared to the computational time required to calculate the quality attributes.

The GA without parameter control took 412 seconds for the 50 000 function evaluations and generated on average 49 non-dominated solutions for the largest component deployment problem (h72 c180) and 15 solutions for the smallest problem (h15 c23). The runs of the BGA took on average 416 seconds, and 89 non-dominated solutions were produced at the end of the optimisation process on average for the largest problem and 12 for the smallest. The GA with the PM technique run on average for 413 seconds and produced 99 solutions for h72 c180 problem and 20 solutions for h15 c23. AP run for 414 seconds and produced 49 non-dominated solutions for h72 c180 and 16 solutions for h15 c23. DMAB had a similar running time (415), and produced 63 non-dominated solutions for h72 c180 and 17 solutions for h15 c23. In general, PM was the fastest parameter control technique, which is mainly attributed to the simple calculations involved in the estimation of the optimal parameter values. However, the difference in run-time is not very significant between the different optimisation schemes. Further optimisations of the code may improve the performance of these methods.

Table 12: The means and standard deviations of hypervolume indicators for the 50 runs.

Redundancy allocation										
	Mean					**Standard deviation**				
Problem	BGA	PM	DMAB	AP	GA	BGA	PM	DMAB	AP	GA
c15	**0.839**	0.829	0.829	0.827	0.831	9.1e-04	1.5e-03	1.3e-03	1.5e-03	1.3e-03
c33	**0.943**	0.937	0.937	0.937	0.935	6.9e-04	3.2e-04	3.4e-04	2.8e-04	3.5e-04
c67	**0.755**	0.739	0.740	0.740	0.738	1.0e-03	1.3e-03	1.2e-03	1.2e-03	1.3e-03
c89	**0.781**	0.765	0.770	0.770	0.770	1.1e-03	1.5e-03	1.4e-03	1.4e-03	1.5e-03
c123	**0.922**	0.915	0.916	0.915	0.913	6.4e-04	4.1e-04	3.6e-04	3.9e-04	5.2e-04
c150	**0.807**	0.799	0.797	0.797	0.793	5.8e-04	1.1e-03	9.2e-04	1.1e-03	1.2e-03
c185	**0.918**	0.911	0.909	0.910	0.909	5.5e-04	4.8e-04	4.2e-04	4.6e-04	4.9e-04
c220	**0.837**	0.826	0.828	0.827	0.825	5.3e-04	9.4e-04	9.1e-04	9.8e-04	9.5e-04
Component deployment										
	Mean					**Standard deviation**				
Problem	BGA	PM	DMAB	AP	GA	BGA	PM	DMAB	AP	GA
h15c23	**0.907**	0.898	0.899	0.900	0.898	7.6e-04	4.4e-04	4.4e-04	4.3e-04	3.9e-04
h15c32	**0.728**	0.714	0.713	0.717	0.708	9.1e-04	1.3e-03	1.2e-03	1.0e-03	1.3e-03
h33c51	**0.921**	0.917	0.917	0.916	0.912	3.4e-04	3.6e-04	2.6e-04	4.1e-04	4.3e-04
h33c67	**0.905**	0.901	0.901	0.901	0.896	4.9e-04	4.7e-04	3.7e-04	4.2e-04	4.8e-04
h55c76	**0.645**	0.637	0.638	0.640	0.628	8.8e-04	9.9e-04	8.6e-04	8.6e-04	8.4e-04
h55c120	**0.577**	0.570	0.571	0.570	0.556	7.2e-04	9.5e-04	8.4e-04	8.0e-04	1.1e-03
h72c130	**0.845**	0.837	0.836	0.836	0.834	4.6e-04	8.6e-04	7.8e-04	7.2e-04	8.4e-04
h72c180	**0.621**	0.604	0.605	0.603	0.592	7.3e-04	9.9e-04	9.7e-04	1.0e-03	9.2e-04

Results from the experimental evaluation of the different optimisation schemes are presented in Table 12. The means and standard deviations of the 50 runs are reported for each optimisation scheme and problem instance. The mean performance of the Genetic Algorithm with the Bayesian parameter control



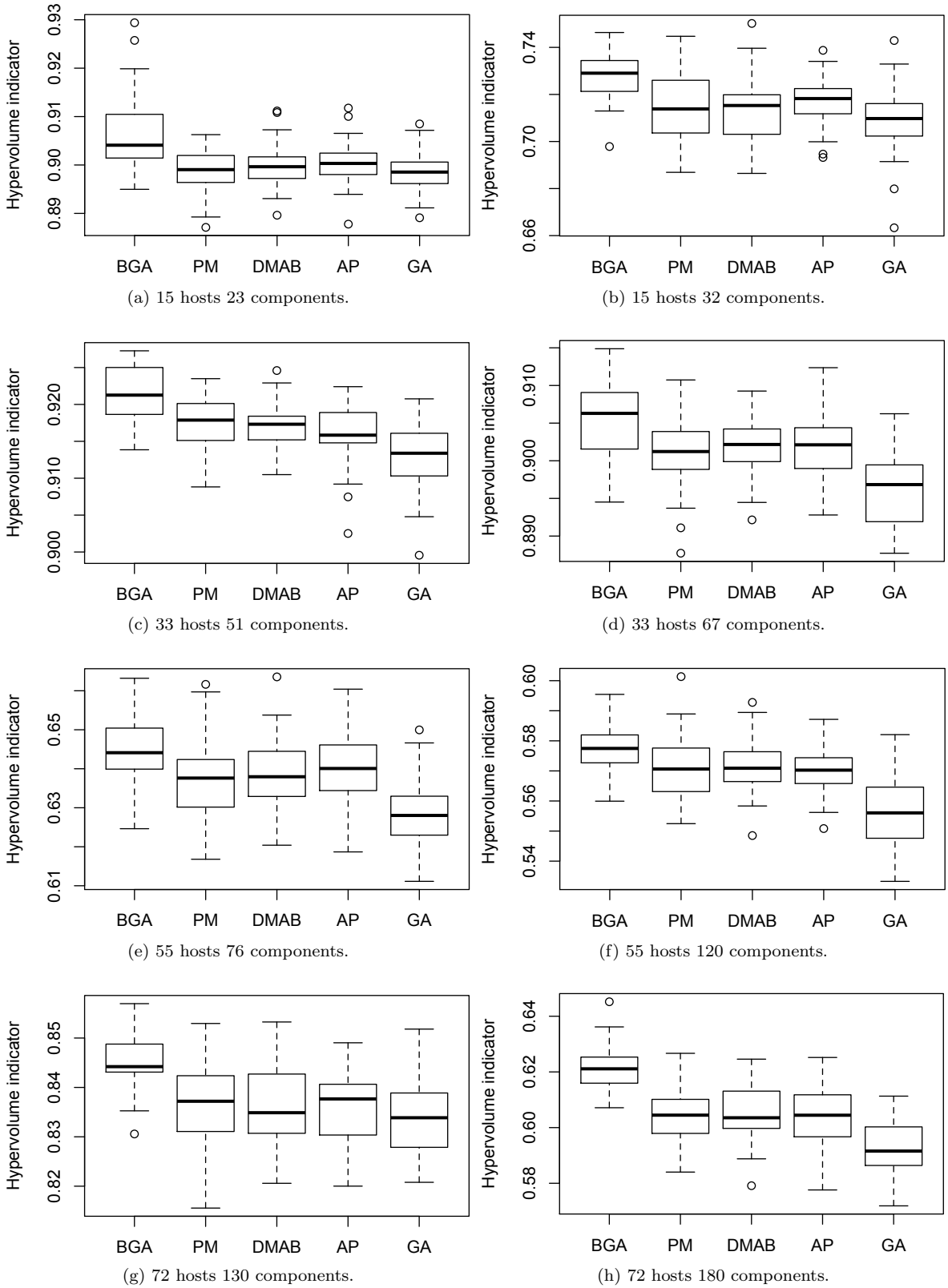
(a) 15 components. (b) 33 components.

(c) 67 components. (d) 89 components.

(e) 123 components. (f) 150 components.

(g) 185 components. (h) 220 components.

Fig. 15: Boxplots of the 50 trials of the optimisation schemes used for the redundancy allocation problems.



(a) 15 hosts 23 components.

(b) 15 hosts 32 components.

(c) 33 hosts 51 components.

(d) 33 hosts 67 components.

(e) 55 hosts 76 components.

(f) 55 hosts 120 components.

(g) 72 hosts 130 components.

(h) 72 hosts 180 components.

Fig. 16: Boxplots of the 50 trials of the optimisation schemes used for the component deployment problems.

Table 13: Kolmogorov-Smirnov test values and effect size of hypervolume indicators for the 50 runs of BGA vs. PM, BGA vs. DMAB, BGA vs. AP, and BGA vs. GA.

Redundancy allocation								
	BGA vs. PM		BGA vs. DMAB		BGA vs. AP		BGA vs. GA	
Problem	d-value	p-value	d-value	p-value	d-value	p-value	d-value	p-value
c15	0.388	0.001	0.449	0.000	0.388	0.001	0.347	0.004
c33	0.429	0.000	0.449	0.000	0.449	0.000	0.592	0.000
c67	0.510	0.000	0.489	0.000	0.469	0.000	0.531	0.000
c89	0.510	0.000	0.469	0.000	0.429	0.000	0.367	0.002
c123	0.469	0.000	0.449	0.000	0.449	0.000	0.633	0.000
c150	0.531	0.000	0.551	0.000	0.489	0.000	0.653	0.000
c185	0.489	0.000	0.633	0.000	0.592	0.000	0.612	0.000
c220	0.633	0.000	0.510	0.000	0.489	0.000	0.673	0.000
Problem	Cohen's d	e-size	Cohen's d	e-size	Cohen's d	e-size	Cohen's d	e-size
c15	8.151	0.971	8.612	0.974	9.604	0.979	6.783	0.959
c33	10.887	0.983	11.162	0.984	10.522	0.982	14.755	0.991
c67	12.861	0.988	12.434	0.987	12.126	0.987	13.617	0.989
c89	11.266	0.984	8.368	0.973	8.217	0.972	7.882	0.969
c123	13.196	0.989	10.195	0.981	12.611	0.988	15.627	0.992
c150	8.064	0.971	11.807	0.986	11.149	0.984	14.398	0.990
c185	12.558	0.988	17.779	0.994	15.933	0.992	16.583	0.993
c220	13.603	0.989	10.998	0.984	11.854	0.986	14.334	0.990
Component deployment								
	BGA vs. PM		BGA vs. DMAB		BGA vs. AP		BGA vs. GA	
Problem	d-value	p-value	d-value	p-value	d-value	p-value	d-value	p-value
h15c23	0.612	0.000	0.592	0.000	0.489	0.000	0.673	0.000
h15c32	0.531	0.000	0.592	0.000	0.489	0.000	0.694	0.000
h33c51	0.449	0.000	0.633	0.000	0.551	0.000	0.775	0.000
h33c67	0.408	0.000	0.429	0.000	0.429	0.000	0.674	0.000
h55c76	0.408	0.000	0.408	0.000	0.286	0.000	0.755	0.000
h55c120	0.388	0.000	0.367	0.000	0.429	0.000	0.775	0.000
h72c130	0.633	0.000	0.674	0.000	0.735	0.000	0.735	0.000
h72c180	0.714	0.000	0.653	0.000	0.694	0.000	0.979	0.000
Problem	Cohen's d	e-size	Cohen's d	e-size	Cohen's d	e-size	Cohen's d	e-size
h15c23	13.192	0.989	11.583	0.985	10.678	0.983	13.825	0.990
h15c32	11.635	0.986	12.882	0.988	10.718	0.983	16.539	0.993
h33c51	11.692	0.986	14.147	0.990	14.133	0.990	21.741	0.996
h33c67	8.088	0.971	8.002	0.970	7.429	0.966	18.702	0.994
h55c76	8.209	0.972	7.687	0.968	5.514	0.940	19.599	0.995
h55c120	8.052	0.971	8.254	0.972	9.271	0.978	22.611	0.996
h72c130	12.376	0.987	14.274	0.990	15.087	0.991	15.765	0.992
h72c180	19.302	0.995	18.666	0.994	19.652	0.995	34.699	0.998

strategy (BGA) is consistently above the average of the other optimisation schemes, and clearly indicates a significant difference between the result groups of BGA and the benchmarks. The standard deviations of the different optimisation schemes are quite similar.

For a better understanding of the behaviour of the algorithms, the results from the 50 runs of each optimisation scheme and problem instance are depicted as box-plots. Figure 15 depicts the results for the redundancy allocation problem, whereas Figure 16 shows the results for the component deployment problem. The performance genetic algorithm without parameter control (GA) is relatively good for the smallest problem instance of the redundancy allocation problem shown in Figure 15a. It's performance is higher than the benchmark parameter control methods, but is outperformed by BGA, although the difference is small. The search space of instance c15 is relatively small, and the problem can be solved to high quality with pre-tuned parameter values. For larger and more complex search spaces, the GA with pre-tuned parameter values fails at finding high-quality results. Different results are observed for the component deployment problem (c.f. Figure 16), where the GA without parameter control is outperformed by all other methods in all problem instances. The Genetic Algorithm with Bayesian Parameter Control is again the best-performing algorithm in all problem instances. The difference in performance is higher

for the larger problems (h72 c130 and h72 c180), which not only have a large search space, but are also more constrained. BGA is very successful in finding high-quality solutions for these instances.

To check for a statistical significance of the results we used the Kolmogorov-Smirnov non-parametric test (Pettitt and Stephens, 1977). The Adaptive Genetic Algorithm was compared against NSGA-II with pre-tuned parameter values and the other adaptive parameter control schemes (PM, AP and DMAB), with a null hypothesis of no difference between the performances (BGA vs. PM, BGA vs. DMAB, and BGA vs. AP). The 50 values of the hypervolume indicators of the repeated trials for each problem instance were submitted to the KS analysis and results are shown in Tables 13. All KS tests, used for establishing differences between independent datasets under the assumption that they are not normally distributed, result in a rejection of the null hypothesis with a minimum d-value of 0.286 at a 100% confidence level. Hence we conclude that the superior performance of the Genetic Algorithm with Bayesian Parameter Control is statistically significant.

The differences in the results are more pronounced in the component allocation problem, where the d-values are higher compared to the redundancy allocation problem. The search space of the component deployment problems is larger compared to the redundancy allocation problem, hence we can conclude that the benefit of using BGA is more evident for more difficult problems. The KS-test also shows that the differences in performance are higher in the trials using the bigger instances of the component deployment and redundancy allocation problems, where d-values are usually higher than the smaller instances. While the d-values for c15 are in the range $0.347 - 0.449$, in c220 the d-value are in the range 0.489-0.673. Similarly, the d-values for the component deployment instance h15 c23 are lower (in the range 0.489-0.673) that the d-values of h72 c180, which are in the range $[0.653, 0979]$. It can be concluded that the least benefit BGA provides for the smallest instance of the redundancy allocation problem (c15). This is an 'easier' instance to solve, hence the algorithm performance can be expected to be more robust to parameter settings. Nonetheless, the Kolmogorov-Smirnov test (c.f. Table 13) finds a significantly superior performance of BGA compared to state-of-the-art parameter control methods on all 18 problem instances.

Since statistical significance depends on the sample size, we also compute the effect size for each comparison (BGA vs. PM, BGA vs. DMAB, and BGA vs. AP), as shown in Table 12. The effect of the sample size is measured using the Cohen's d estimation (Cohen, 1988), which considers the pooled standard deviation. In reporting the effect size, we follow the guidelines proposed by Cohen (1988): a 'small' effect size is 0.2, a 'medium' effect size is 0.5, and a 'large' effect size is 0.8. In essence, the effect size indicates the number of standard deviations difference between the means of the samples. The effect size for all problem instances was above 0.9. As a results, it can be concluded that the difference in the performance of the optimisation schemes is meaningful.

7.4 Threats to Validity

The validity of the presented experiments can be questioned on the grounds that the method may only be effective in certain problem instances. The new parameter control method presented outperformed other methods in the 8 problems instances, and there is a chance that the approach may not perform in the same way for other problems, and when applied to the parameter adaptation of other optimisation algorithms. In the design of experiment, we aimed at reducing this threat by creating problem instances of different sizes and characteristics. Instead of manually setting specific problem properties, we developed a problem generator integrated in ArcheOpterix (Aleti et al., 2009a). As a result, the experiments set themselves apart from an instance-specific setting to a broader applicability.

Another important threat is the stochastic behaviour of Genetic Algorithms, which may lead to different results for the same problem instance. This threat was reduced by having multiple runs (50) for each optimisation method and problem instance. Statistical tests were applied to check for statistical significance in the results. To measure the strength of the out-performance of the proposed method, the effect size for each experiment was reported.

The experimental results may be affected by the implementation of the algorithms. The is a chance that the code contains errors or bias towards certain methods. To address this problem, we have consulted experienced programmers and followed regular code-review sessions. The implementation was checked for errors several times, and tests were conducted, which we hope have minimised the possibility of misleading results in the experiments.

8 Discussion and Future Work

The adaptive parameter control introduced in this paper was used to adjust parameters of a Genetic Algorithm. The performance of the GA with the adaptive parameter control method improved significantly compared to the GA with pre-tuned parameter values. The method can easily be adapted to other search algorithms, such as Ant Colony Optimisation, Simulated Annealing and other stochastic local search methods, which will be investigated in future work. The contribution presented in this paper is directly related to the research efforts in Search Based Software Engineering (Harman, 2007a; Harman et al., 2012a). There is an emerging interest in using search method for software engineering problems in academia (Aleti et al., 2013; Harman, 2007b; McMinn, 2004; Mantere and Alander, 2005) and industry (Afzal et al., 2010; Yoo et al., 2011; Comford et al., 2003). All these methods would benefit from the adaptive parameter control presented in this paper.

One of the main benefits of the proposed method is that it removes the task of tuning parameter values of a search method to a particular problem, which can be a time-consuming and laborious task. The method would be especially useful for practitioners, who usually have little expertise in the Artificial Intelligence field, and require guidance in the application of search algorithms to their particular problems. There are few parametrisation guidelines for the interdisciplinary users of these algorithms, and the available information is often conflicting. For instance, recommended values for crossover rate can vary between 0.6 (De Jong, 1995) to 0.95 (Grefenstette, 1986), or any value in the range $[0.75, 0.95]$ (Schaffer et al., 1989), whereas recommended mutation rate values vary between 0.001 (DeJong, 1975) and 0.05 (Grefenstette, 1986). The conflicting guidelines are mainly due to the fact that different parameter values are optimal for different problems (Mitchell, 1996), hence the parametrisation becomes an optimisation problem in its own right. But most importantly, it is has been empirically proven that different parameter settings are also required at different stages of the optimisation process (Hesser and Manner, 1991; Cervantes and Stephens, 2009; Thierens, 2002), which was also confirmed by the results of our experiments. The GA with pre-tuned parameter values performed consistently worse than the GA with adaptive parameter control. Adapting algorithm parameters during the search process would be beneficial for other problems in software engineering, hence the application of the adaptive parameter control to the area of SBSE is a priority.

9 Conclusion

This paper presented an adaptive approach for controlling parameter values of Genetic Algorithms. The process of adapting parameter values is performed during the optimisation process. The approach was applied to a case study from the automotive industry and a set of instances from the component deployment and redundancy allocation problems. Architecture optimisation in embedded systems is an example of optimisation problems that are faced in the industry. The case study and the experimental evaluation showed that the approach introduced in this paper achieves better results compared to tuned parameter settings, and out-performed 3 state-of-the-art adaptive parameter control methods. Most importantly, the reduction in the number of the GA parameters that have to be set by a practitioner facilitates the transfer of GAs into industrial settings, where practitioners do not have any knowledge on how to tune algorithm parameters.

Acknowledgements

We wish to acknowledge Monash University for the use of their Nimrod software in this work. The Nimrod project has been funded by the Australian Research Council and a number of Australian Government agencies, and was initially developed by the Distributed Systems Technology. This research was supported under Australian Research Council's Discovery Projects funding scheme (project number DE140100017). We would like to thank several anonymous reviewers who helped improve the quality of this work.

References

Afzal, W. and Torkar, R. (2011). On the application of genetic programming for software engineering predictive modeling: A systematic review. *Expert Systems with Applications*, 38(9):11984–11997.

Afzal, W., Torkar, R., Feldt, R., and Wikstrand, G. (2010). Search-based prediction of fault-slip-through in large software projects. In *Second International Symposium on Search Based Software Engineering*, pages 79–88.

Åkerholm, M., Fredriksson, J., Sandström, K., and Crnkovic, I. (2004). Quality attribute support in a component technology for vehicular software. In *Fourth Conference on Software Engineering Research and Practice*.

Aleti, A., Björnander, S., Grunske, L., and Meedeniya, I. (2009a). ArcheOpterix: An extendable tool for architecture optimization of AADL models. In *Model-based Methodologies for Pervasive and Embedded Software*, pages 61–71. IEEE Digital Libraries.

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

Aleti, A., Grunske, L., Meedeniya, I., and Moser, I. (2009b). Let the ants deploy your software - an ACO based deployment optimisation strategy. In *Automated Software Engineering*, pages 505–509. IEEE Digital Library.

Aleti, A. and Moser, I. (2011). Predictive parameter control. In *Genetic and Evolutionary Computation Conference, Proceedings*, pages 561–568.

Aleti, A., Moser, I., and Mostaghim, S. (2012). Adaptive range parameter control. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE.

Angeline, P. J. (1995). Adaptive and self-adaptive evolutionary computations. In Palaniswami, M. and Attikiouzel, Y., editors, *Computational Intelligence: A Dynamic Systems Perspective*, pages 152–163. IEEE Press.

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press.

Bäck, T. (2001). Introduction to the special issue: Self-adaptation. *Evolutionary Computation*, 9(2):iii–iv.

Bartz-Beielstein, T., Lasarczyk, C., and Preuss, M. (2005). Sequential parameter optimization. In *IEEE Congress on Evolutionary Computation*, pages 773–780. IEEE.

Broy, M. (2006). Challenges in automotive software engineering. In *International Conference on Software Engineering (ICSE'06)*, pages 33–42. ACM.

Butts, K., Bostic, D., Chutinan, A., Cook, J., Milam, B., and Wang, Y. (2001). Usage scenarios for an automated model compiler. *Lecture Notes in Computer Science*, 2211:66–79.

Calinescu, R. and Kwiatkowska, M. (2009). Using quantitative analysis to implement autonomic IT systems. In *International Conference on Software Engineering, ICSE*, pages 100–110. IEEE.

Cervantes, J. and Stephens, C. R. (2009). Limitations of existing mutation rate heuristics and how a rank GA overcomes them. *IEEE Trans. Evolutionary Computation*, 13(2):369–397.

Chern, M.-S. (1992). On the computational complexity of reliability redundancy allocation in a series system. *Operations Research Letters*, 11(5):309315.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences*.

Coit, D. W. and Smith, A. E. (1996). Reliability optimization of series-parallel systems using a genetic algorithm. *IEEE Transactions on Reliability*, 45(2):254–260.

Comford, S., Feather, M., Dunphy, J., Salcedo, J., and Menzies, T. (2003). Optimizing spacecraft design optimization engine development: progress and plans. In *Proceedings of the Aerospace Conference*, volume 8, pages 3681–3690. IEEE.

Corne, D., Oates, M. J., and Kell, D. B. (2002). On fitness distributions and expected fitness gain of mutation rates in parallel evolutionary algorithms. In *Parallel Problem Solving from Nature*, volume 2439 of *Lecture Notes in Computer Science*, pages 132–141. Springer-Verlag.

Cuenot, P., Chen, D.-J., Gérard, S., Lönn, H., Reiser, M.-O., Servat, D., Sjöstedt, C.-J., Kolagari, R. T., Törngren, M., and Weber, M. (2007). Managing complexity of automotive electronics using the EAST-ADL. In *ICECCS*, pages 353–358. IEEE Computer Society.

Davis, L. (1989). Adapting operator probabilities in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 70–79. Morgan Kaufman.

De Jong, K. A. (1995). *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan.

Deb, K. and Beyer, H.-G. (2001). Self-adaptive genetic algorithms with simulated binary crossover. *Evolutionary Computation*, 9(2):197–221.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Evolutionary Computation*, 6:182–197.

DeJong, K. (2007). Parameter setting in EAs: a 30 year perspective. In *Parameter Setting in Evolutionary Algorithms*, volume 54 of *Studies in Computational Intelligence*, pages 1–18. Springer.

DeJong, K. A. (1975). *Analysis of Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, The University of Michigan.

Eiben, A. E., Hinterding, R., and Michalewicz, Z. (2007). Parameter control in evolutionary algorithms. *IEEE Transations on Evolutionary Computation*, 3(2):124–141.

Eiben, A. E. and Schut, M. C. (2008). New ways to calibrate evolutionary algorithms. In Siarry, P. and Michalewicz, Z., editors, *Advances in Metaheuristics for Hard Optimization*, Natural Computing Series, pages 153–177. Springer.

Eiben, A. E. and Smit, S. K. (2011). Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31.

Farmani, R. and Wright, J. A. (2003). Self-adaptive fitness formulation for constrained optimization. *IEEE Transactions on Evolutionary Computation*, 7(5):445–455.

Fialho, Á., Schoenauer, M., and Sebag, M. (2009). Analysis of adaptive operator selection techniques on the royal road and long k-path problems. In *Genetic and Evolutionary Computation Conference, Proceedings*, pages 779–786. ACM.

Fogarty, T. C. (1989). Varying the probability of mutation in the genetic algorithm. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 104–109. Morgan Kaufman.

Fredriksson, J., Nolte, T., Nolin, M., and Schmidt, H. (2007). Contract-based reusable worst-case execution time estimate. In *The International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 39–46.

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

Goševa-Popstojanova, K. and Trivedi, K. S. (2001). Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2-3):179–204.

Grefenstette, J. J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-16(1):122–128.

Grunske, L. (2006). Identifying "good" architectural design alternatives with multi-objective optimization strategies. In *International Conference on Software Engineering*, pages 849–852. ACM.

Harman, M. (2007a). The current state and future of search based software engineering. In Briand, L. C. and Wolf, A. L., editors, *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering*, pages 342–357.

Harman, M. (2007b). Search based software engineering for program comprehension. In *15th International Conference on Program Comprehension (ICPC 2007)*. IEEE. Invited paper.

Harman, M., Burke, E. K., Clark, J. A., and Yao, X. (2012a). Dynamic adaptive search based software engineering. In *International Symposium on Empirical Software Engineering and Measurement*, pages 1–8. ACM.

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

Heinecke, H., Schnelle, K. P., Fennel, H., Bortolazzi, J., Lundh, L., Leflour, J., Mat, J.-L., Nishikawa, K., and Scharnhorst, T. (2004). Automotive open system architecture-an industry-wide initiative to manage the complexity of emerging automotive E/E architectures. In *Convergence transportation electronics Association*. ACM.

Heiner, G. and Thurner, T. (1998). Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In *International Symposium on Fault-Tolerant Computing*, pages 402–407.

Hesser, J. and Manner, R. (1991). Towards an optimal mutation probability for genetic algorithms. *Lecture Notes in Computer Science*, 496:23–32.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan.

Hong, T.-P., Wang, H.-S., and Chen, W.-C. (2000). Simultaneously applying multiple mutation operators in genetic algorithms. *Journal of Heuristics*, 6(4):439–455.

Igel, C. and Kreutz, M. (2003). Operator adaptation in evolutionary computation and its application to structure optimization of neural networks. *Neurocomputing*, 55(1-2):347–361.

ISO/IEC (2000). IEEE International Standard 1471 2000 - Systems and software engineering - Recommended practice for architectural description of software-intensive systems.

Jhumka, A., Hiller, M., and Suri, N. (2002). Component-based synthesis of dependable embedded software. In Damm, W. and Olderog, E.-R., editors, *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT*, volume 2469, pages 111–128.

Julstrom, B. A. (1995). What have you done for me lately? Adapting operator probabilities in a steady-state genetic algorithm. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 81–87, San Francisco, CA. Morgan Kaufmann.

Kitchenham, B., Pickard, L., and Pfleeger, S. L. (1995). Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62.

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

Kulturel-Konak, S. and Coit, A. E. S. D. W. (2003). Efficiently solving the redundancy allocation problem using tabu search. *IIE Transactions*, 35(6):515–526.

Kulturel-Konak, S., Coit, D. W., and Baheranwala, F. (2008). Pruned pareto-optimal sets for the system redundancy allocation problem based on multiple prioritized objectives. *Journal of Heuristics*, 14(4):335–357.

Liang, Y.-C. and Smith, A. E. (1999). An ant system approach to redundancy allocation. In *Congress on Evolutionary Computation*, pages 1478–1484. IEEE.

Lukasiewycz, M., Glaß, M., Haubelt, C., and Teich, J. (2008). Efficient symbolic multi-objective design space exploration. In *ASP-DAC 2008*, pages 691–696. IEEE.

Malek, S. (2007). *A User-Centric Approach for Improving a Distributed Software System's Deployment Architecture*. PhD thesis, Faculty of the graduate schools, University of Southern California.

Malek, S., Mikic-Rakic, M., and Medvidovic, N. (2005). A decentralized redeployment algorithm for improving the availability of distributed systems. In *Component Deployment*, volume 3798 of *Lecture Notes in Computer Science*, pages 99–114. Springer.

Mantere, T. and Alander, J. T. (2005). Evolutionary software engineering, a review. *Applied Soft Computing*, 5(3):315–331.

Martens, A. and Koziolek, H. (2009). Automatic, model-based software performance improvement for component-based software designs. In *6th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA)*. Elsevier.

McMinn, P. (2004). Search-based software test data generation: a survey. *Softw. Test, Verif. Reliab*, 14(2):105–156.

Medvidovic, N. and Malek, S. (2007). Software deployment architecture and quality-of-service in pervasive environments. In *Workshop on the Engineering of Software Services for Pervasive Environements*, pages 47–51. ACM.

Meedeniya, I., Buhnova, B., Aleti, A., and Grunske, L. (2011). Reliability-driven deployment optimization for embedded systems. *Journal of Systems and Software*, 84(5):835–846.

Mikic-Rakic, M., Malek, S., Beckman, N., and Medvidovic, N. (2004). A tailorable environment for assessing the quality of deployment architectures in highly distributed settings. In *Component Deployment*, volume 3083 of *LNCS*, pages 1–17. Springer.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. Complex Adaptive Systems. MIT, Cambridge.

Nadi, F. and Khader, A. T. A. (2011). A parameter-less genetic algorithm with customized crossover and mutation operators. In *Genetic and Evolutionary Computation Conference*, pages 901–908. ACM.

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

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Pettitt, A. N. and Stephens, M. A. (1977). The kolmogorov-smirnov goodness-of-fit statistic with discrete and grouped data. *Technometrics*, 19(2):205–210.

Pretschner, A., Broy, M., Krüger, I. H., and Stauner, T. (2007). Software engineering for automotive systems:Aroadmap. In *International Conference on Software Engineering*, pages 55–71.

Runeson, P., Hst, M., Rainer, A., and Regnell, B. (2012). John Wiley and Sons, Inc.

Schaffer, J. D., Caruana, R. A., Eshelman, L. J., and Das, R. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Conference on Genetic Algorithms*, pages 51–60. Morgan Kaufman.

Scharnhorst, T., Heinecke, H., Schnelle, K. P., Fennel, H., Bortolazzi, J., Lundh, L., Heitkamper, P., Leflour, J., Mate, J.-L., and Nishikawa, K. (2005). Autosar challenges and achievements. In *VDI Berichte Nr. 1907*, pages 395–408.

Schlierkamp-Voosen, D. and Mühlenbein, H. (1994). Strategy adaptation by competing subpopulations. *Lecture Notes in Computer Science*, 866:199–208.

Sharma, V. S., Jalote, P., and Trivedi, K. S. (2005). Evaluating performance attributes of layered software architecture. In *Symposium on Component-Based Software Engineering*, volume 3489 of *LNCS*, pages 66–81. Springer.

Sharma, V. S. and Trivedi, K. S. (2007). Quantifying software performance, reliability and security: An architecture-based approach. *Journal of Systems and Software*, 80(4):493–509.

Shatz, S. M., Wang, J.-P., and Goto, M. (1992). Task allocation for maximizing reliability of distributed computer systems. *IEEE Transactions on Computers*, 41(9):1156–1168.

Smit, S. K. and Eiben, A. E. (2009). Comparing parameter tuning methods for evolutionary algorithms. In *IEEE Congress on Evolutionary Computation*, pages 399–406. IEEE.

Smith, J. and Fogarty, T. C. (1996). Self adaptation of mutation rates in a steady state genetic algorithm. In *International Conference on Evolutionary Computation*, pages 318–323.

Thierens, D. (2002). Adaptive mutation rate control schemes in genetic algorithms. In *Proceedings of the 2002 Congress on Evolutionary Computations*, pages 980–985. IEEE.

Thierens, D. (2005). An adaptive pursuit strategy for allocating operator probabilities. In *Genetic and Evolutionary Computation Conference*, pages 1539–1546. ACM.

Tuson, A. and Ross, P. (1998). Adapting operator settings in genetic algorithms. *Evolutionary Computation*, 6(2):161–184.

Verner, J. M., Sampson, J., Tosic, V., Bakar, N. A. A., and Kitchenham, B. (2009). Guidelines for industrially-based multiple case studies in software engineering. In *International Conference on Research Challenges in Information Science*, pages 313–324. IEEE.

Yoo, S., Nilsson, R., and Harman, M. (2011). Faster fault finding at google using multi objective regression test optimisation. In *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

Zhang, Y., Finkelstein, A., and Harman, M. (2008). Search based requirements optimisation: Existing work and challenges. 5025:88–94.

Zitzler, E., Brockhoff, D., and Thiele, L. (2007). The Hypervolume Indicator Revisited:On the Design of Pareto-compliant Indicator Via Weighted Integration. In *Evolutionary Multi-Criterion Optimization*, volume 4403 of *LNCS*, pages 862–876. Springer.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and da V. G. Fonseca (2003). Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transaction on Evolutionary Computation*, 7:117–132.