Andrade and Macêdo *Journal of Software Engineering Research and Development* (2015) 3:2
DOI 10.1186/s40411-015-0016-z

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

Assessing the benefits of search-based approaches when designing self-adaptive systems: a controlled experiment

Sandro S Andrade[1,2*†] and Raimundo J de A Macêdo[1†]

*Correspondence: sandros@ufba.br
†Equal contributors
[1]Distributed Systems Laboratory (LaSiD), Federal University of Bahia (UFBa), Institute of Mathematics, Department of Computer Science, Av Adhemar de Barros, s/n, Ondina, 40.170-110 Salvador-BA, Brazil
[2]GSORT Distributed Systems Group, Federal Institute of Education, Science, and Technology of Bahia (IFBa), Department of Computer Science, Av. Araújo Pinho, 39, Canela, 40.110-150 Salvador-BA, Brazil

Abstract

Background: The well-orchestrated use of distilled experience, domain-specific knowledge, and well-informed trade-off decisions is imperative if we are to design effective architectures for complex software-intensive systems. In particular, designing modern self-adaptive systems requires intricate decision-making over a remarkably complex problem space and a vast array of solution mechanisms. Nowadays, a large number of approaches tackle the issue of endowing software systems with self-adaptive behavior from different perspectives and under diverse assumptions, making it harder for architects to make judicious decisions about design alternatives and quality attributes trade-offs. It has currently been claimed that search-based software design approaches may improve the quality of resulting artifacts and the productivity of design processes, as a consequence of promoting a more comprehensive and systematic representation of design knowledge and preventing design bias and false intuition. To the best of our knowledge, no empirical studies have been performed to provide sound evidence of such claim in the self-adaptive systems domain.

Methods: This paper reports the results of a quasi-experiment performed with 24 students of a graduate program in Distributed and Ubiquitous Computing. The experiment evaluated the design of self-adaptive systems using a search-based approach proposed by us, in contrast to the use of a non-automated approach based on architectural styles catalogs. The goal was to investigate to which extent the adoption of search-based design approaches impacts on the effectiveness and complexity of resulting architectures. In addition, we also analyzed the approach's potential for leveraging the acquisition of distilled design knowledge.

Results: Our findings show that search-based approaches can improve the effectiveness of resulting self-adaptive systems architectures and reduce their design complexity. We found no evidence regarding the approach's potential for leveraging the acquisition of distilled design knowledge by novice software architects.

Conclusion: This study contributes to reveal empirical evidence on the benefits of search-based approaches when designing self-adaptive systems architectures. The results presented herein increase our belief that the systematic representation of distilled design knowledge and the adoption of search-based design approaches indeed lead to improved architectures.

Keywords: Self-adaptive systems; Software architecture; Software modeling; Search-based software engineering; Empirical software engineering

1 Background

Modern software-intensive systems are becoming increasingly complex and the fulfillment of requirements for performance, flexibility, dependability, and energy-efficiency in uncertain and dynamic environments is still a quite challenging task (Huebscher and McCann 2008). Elastic data storage services, energy-aware mobile systems, self-tuning databases, and reconfigurable network services are some of the application domains in which self-adaptive mechanisms play a paramount role (Patikirikorala et al. 2012). Such scenarios are usually characterized by incomplete knowledge about user requirements, workloads, and available resources. As a consequence, committing to a particular solution in design time may yield suboptimal architectures, which easily degrade the service when conditions deviate from those previously defined. Establishing the foundations that enable the systematic design, development, and evolution of systems with self-management capabilities has been the focus of many research efforts in areas such as self-adaptive systems, autonomic computing, and artificial intelligence (Salehie and Tahvildari 2009).

A self-adaptive (SA) system continuously monitor its own behavior and its operating environment, adapting itself whenever current conditions prevent it from delivering the expected quality of service (Salehie and Tahvildari 2009). SA systems usually comprise two parts: a managed element and a managing element (Huebscher and McCann 2008). The managed element provides functional services to the user, operating in a potentially dynamic and uncertain environment. The managing system is responsible for adapting the managed element, mostly by using a particular implementation of an adaptation loop. The MAPE-K approach (Kephart and Chess 2003) is a widely accepted reference architecture for adaptation loops. It defines the basic components for the loop's tasks of **M**onitoring, **A**nalyzing, **P**lanning, and **E**xecuting; performed with the support of a **K**nowledge Base.

The many approaches for self-adaptation available nowadays employ different mechanisms for the aforementioned tasks. Reflexive middleware platforms (Ogel et al. 2003), graph grammars (Bruni et al. 2008), intelligent agents (Benyon and Murray 1993), policy-based approaches (Georgas and Taylor 2008), self-organizing structures (Georgiadis et al. 2002), and control theory (Tilbury et al. 2004) are some of the currently adopted underpinnings for enabling self-adaptation. Becoming familiar with the most prominent modeling dimensions for SA systems is crucial for specifying relevant adaptation requirements; making unbiased and well-informed decisions about alternative architectures; and accurately evaluating the resulting system's quality attributes.

Although some previous work have already tackled this issue (Andersson et al. 2009; Brun et al. 2009; de Lemos et al. 2010; Patikirikorala et al. 2012), representing such dimensions by using ill-structured notations – such as natural language, ad-hoc architectural styles catalogs, and informally depicted reference architectures – makes it harder and time-consuming for novice architects to grasp the architectural tactics that lead to particular adaptation quality attributes. As a consequence, effective subtle architectures for an adaptation problem at hand may not be considered because of design bias, limited knowledge about the solution domain, or time constraints. Furthermore, the lack of flexible, expressive, and automated mechanisms for performing the usual decide-design-evaluate cycles hampers the eliciting of relevant insights about quality attributes trade-offs.

Over the past twelve years, Search-Based Software Engineering (SBSE) (Harman et al. 2012) has provided promising approaches for tackling the aforementioned issues in areas

such as requirements engineering, design, testing, and refactoring, just to mention a few. SBSE claims that the majority of issues in such areas are indeed optimization questions and that the software's virtual nature is inherently well suited for search-based optimization (Harman 2010). In particular, substantial work towards search-based software design (Räihä 2010) advocates the benefits of SBSE in finding out subtle effective designs and providing well-informed means to reveal quality attributes trade-offs.

To the best of our knowledge, the first effort in applying search-based approaches to the design of SA systems is that proposed by us in (Andrade and de Araújo Macêdo 2013a,b). In such work, we provide a meta-modeling infrastructure for defining domain-specific design spaces which systematically capture the domain's prominent design dimensions, their associated variation points (alternative solutions), and the architectural changes required to implement each solution. The goal is to support the automated redesign of an initial model, endowing it with additional capabilities from the application domain at hand. Each domain-specific design space entails a set of quality metrics that evaluate each candidate architecture regarding different attributes. We have been using such an approach to enable the automated design of managing elements for initial (non-adaptive) systems such as web servers and MapReduce distributed architectures (Dean and Ghemawat 2008). Since even small input models usually span huge design spaces, we also provide a domain-independent multi-objective optimization engine. Such engine currently relies on the NSGA-II (Non-dominated Sorting Genetic Algorithm II) algorithm (Deb et al. 2002) to find out a set of Pareto-optimal (Deb and Kalyanmoy 2001) candidate architectures. All these solutions represent optimal architectures, differing only in which quality metric they favor.

In (Andrade and de Araújo Macêdom 2014), we report the results of a quasi-experiment which investigated the benefits of search-based approaches when designing SA systems architectures. In this paper, we extend such a report by providing a more accurate description of involved treatments, analysis of resulting data, and threats to validity. The study was performed with 24 students of a graduate program in Distributed and Ubiquitous Computing and evaluated the design of SA systems using our search-based approach, in contrast to using a style-based non-automated approach. The experiment aimed at evaluating the impact of the adopted design method on three dependent variables: the effectiveness and complexity of resulting architectures, as well as the method's potential for leveraging the acquisition of distilled design knowledge by novice architects. All the material used in the experiment is available at http://wiki.ifba.edu.br/tr-ce012014.

The remainder of this paper is organized as follows. Section 2 presents the foundations of SA systems and feedback control. Section 3 describes the automated software architecture design approach proposed by us and adopted as one of the experiment treatments. Section 4 presents an overview of the experiment. Section 5 explains the experiment objects, the hypotheses being investigated, the adopted measurement approach, and the experiment design. In Section 6, we analyze and discuss the experiment results. Threats to validity are identified in Section 7 and related work is discussed in Section 8. Finally, conclusions and venues for future work are presented in Section 9.

2 Self-adaptive systems and feedback control

A SA system is defined in (DARPA 1997) as "a software that evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what

the software is intended to do, or when better functionality or performance is possible". We adopt a slightly broader perspective by defining a SA system as a system which presents some on-line infrastructure which allows moving a specific development process stage (e.g.: design, implementation, or deployment) – usually undertaken off-line – to runtime. Such definition entails from simple adaptive algorithms to more sophisticated solutions such as self-optimization by automated redeployment (Malek et al. 2010) and self-organization enabled by dynamic architectures (Parunak and Brueckner 2011). Feedback control is a well-established technology for handling dynamic electromechanical systems in areas such as avionics, chemical processes, and factory automation. Under such perspective, the use of feedback control considering software applications as the system under control (Tilbury et al. 2004) is still on its early days, albeit the large number of recent work towards this topic (Patikirikorala et al. 2012).

Figure 1 presents the common elements of a feedback control system. The *target system* is a software system with a *system output* – `y(t)` – which represents the quality attribute (e.g.: average service response time or CPU utilization) intended to be controlled. Such attribute is directly influenced by a *system input* signal – `v(t)`, which manipulates, for instance, buffer sizes or the number of threads in a pool. The goal is to retain the system output as close as possible to a *reference input*, which represents the desired service level specified by the administrator. Uncertainties in the operating environment (e.g.: changing workloads or hardware failures) introduce a *disturbance input* signal – `d(t)` – which makes it harder to derive accurate models for system input-output relationships. *Noise input* signals – `n(t)` – produced by sensors with high stochastic sensitivity may further complicate the control goals. Dealing with unmodeled and unforeseen disturbances and noises has motivated the idea where the *measured output* – `m(t)=y(t)+n(t)` – is fed back to the controller. By calculating how much the measured output deviates from the reference input (*control error* – `e(t)`), the feedback controller makes use of some specific control law to decide about the *control input* signal – `u(t)` – to be applied in the target system. A transducer is commonly used in cases demanding unit conversion and/or delay handling.

It is worth mentioning that, in contrast to frameworks for steady-state analysis such as queue theory, control theory provides the means to design controllers that systematically exhibit particular behavior for both steady-state and transient responses. That allows for characterizing the resulting system (controller + target system) in terms of the so-called SASO properties (Tilbury et al. 2004): **S**tability, **A**ccuracy, **S**ettling time, and **O**vershoot.

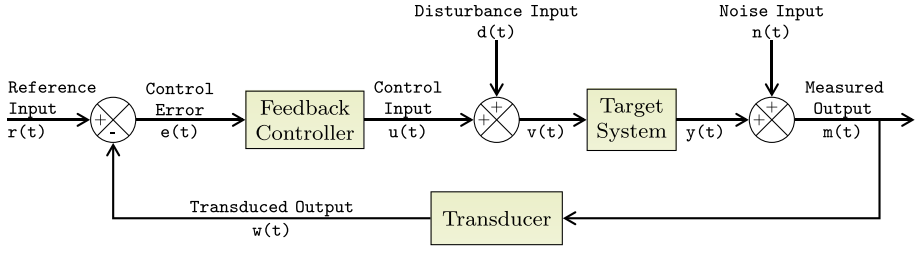

Figure 1 Basic elements of a feedback control loop. The feedback controller aims at retaining the target system's output `y(t)` as close as possible to the reference input `r(t)`, irrespective of the presence of unforeseen disturbance and noise signals.

Among several definitions of stability available today (Slotine and Li 1991), one widely used in feedback control is that of BIBO-stability (Hayes 2011). A system is BIBO-stable if for any bounded input signal $v(t)$ the output $y(t)$ is bounded.

Being the resulting system stable, the remaining SASO properties can be investigated. As depicted in Figure 2a, the smaller the steady-state error e_{ss} (difference between the reference input and measured output), the more accurate is the resulting system. The settling time K_s is the time elapsed from the change in input to when the measured output is within some variation range (usually 2%) of its steady-state value. Finally, the maximum overshoot M_p is the normalized maximum amount by which the system output exceeds its steady-state value.

The systematic design of control architectures which exhibit intentionally chosen values of accuracy, settling time and overshoot is imperative if we are to conceive effective self-adaptive systems. Disregarding such an aspect may lead to over/under provisioning of resources (due to inaccurate convergence), violations of service level agreements (due to slower responses), or excessive use of resources during transient response as a consequence of large overshoots. Figure 2b depicts the step response (dynamics exhibited by the target system when the reference input changes from 0 to 1) for systems with different controllers. The controller 3 presents an ideal response, with no overshoot, high accuracy, and small settling time.

A large body of knowledge regarding control laws and methods for designing controllers is currently available (Patikirikorala et al. 2012). Currently adopted control-theoretic approaches for endowing systems with self-adaptation capabilities include the use of PID control, state-space models, MIMO (Multiple-Input Multiple-Output) control, gain scheduling, self-tuning regulators, fluid flow analysis, and fuzzy control (Tilbury et al. 2004). As a consequence, designing effective architectures for SA systems requires architects become familiar with the intricacies of both the problem space (so that accurate and realistic self-adaptation requirements can be elicited) and solution space (in order to adopt the most effective adaptation strategy/mechanism for the problem at hand). That involves deciding on self-adaptation goals; system and environment monitoring mechanisms; measurement noises and uncertainties; unanticipated/unforeseen adaptations; diverse control robustness degrees; change enacting mechanisms; and adaptation

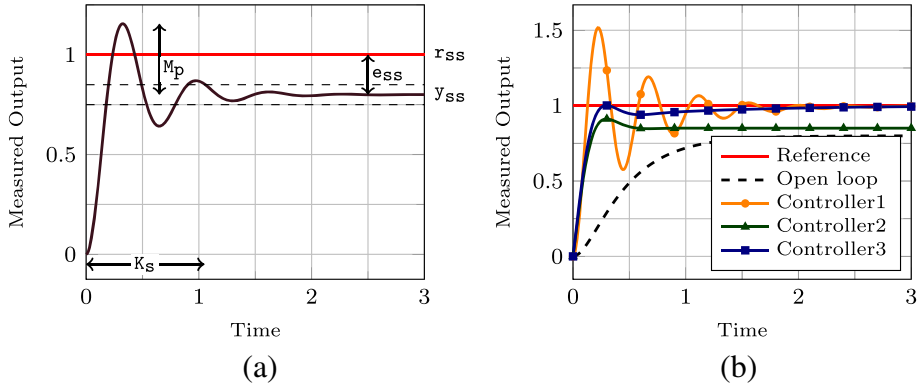

Figure 2 Feedback control properties (a) and step response of systems with different controllers (b). Properties such as settling time (K_s), steady-state error (e_{ss}), and overshoot (M_p) are commonly affected by the control law and tuning techniques chosen for the feedback controller managing the target system.

temporal predictability, just to mention a few (Andersson et al. 2009; Brun et al. 2009; Patikirikorala et al. 2012).

3 Searching for effective self-adaptive systems architectures

A number of efforts from the software engineering for SA systems community (de Lemos et al. 2010) have addressed the issue of providing principled engineering approaches for such an application domain, leveraging the systematic capture of design knowledge and enabling the early reasoning of self-adaptation quality attributes. In previous work (Andrade and de Araújo Macêdo 2013a,b), we presented an infrastructure for systematically representing distilled architecture design knowledge for a given application domain (design space), along with a domain-independent architecture optimization engine as the underlying mechanism for explicitly eliciting design trade-offs (conflicting quality attributes). Such an infrastructure, depicted in Figure 3, provides the underpinnings of our search-based approach for designing self-adaptive systems architectures. The ultimate goal is to support the automatic extension of an initial UML (Unified Modeling Language) model – describing the managed element – with new architectural elements (which implements the managing system), searching for those solutions that exhibit desired control properties.

A concrete design space and its quality attributes are specified by experts once per n domain (*design space inception* stage) by using a modeling language – namely DuSE – we have designed for such a purpose. A supporting UML profile is also defined for that domain, enabling the annotations that drive the automated design process. A *design space* (e.g.: for networked and concurrent systems) is defined as a set of *n design dimensions*

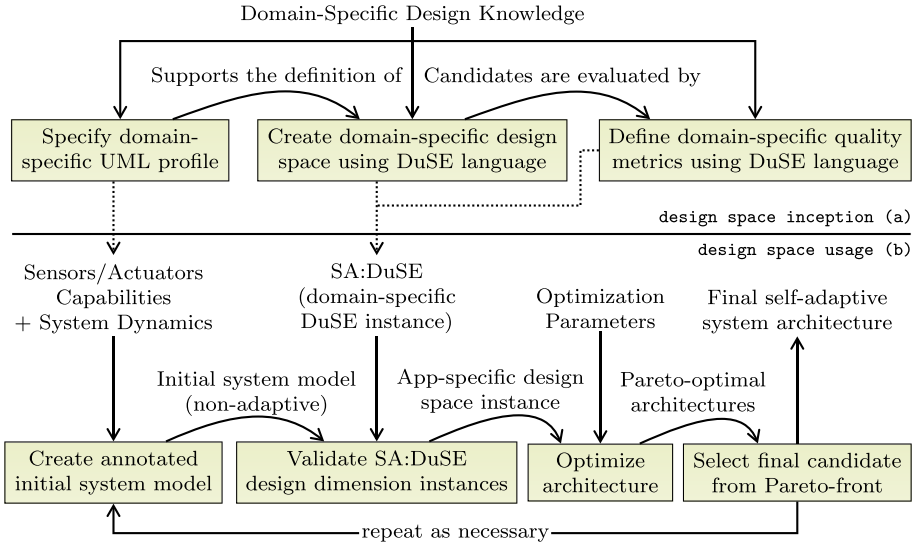

Figure 3 Overview of our architecture design approach. In the *design space inception* stage **(a)**, domain experts use the DuSE's constructs for specifying a domain-specific design space (degrees of freedom, their corresponding alternative solutions, and the architecture extensions required to implement each solution). Henceforth – in the *design space usage* stage **(b)** – architects submit initial models to the optimization engine, which searches for those (near-)optimal extensions revealing design trade-offs. A particular design space instance – SA:DuSE – enables the use of such an infrastructure in the self-adaptive systems domain.

representing specific design concerns in such a domain (e.g.: concurrency strategy and event dispatching model).

Definition 1. *A **design space** is a tuple $ds=\langle DD, QM, P\rangle$, where DD is a non-empty totally ordered set of design dimensions, QM is a non-empty totally ordered set of quality metrics, and P is the accompanying UML profile for such an design space.*

Each design dimension entails a set of *variation points*, representing alternative solutions for such a concern (e.g.: leader-followers or half-sync/half-async; for the concurrency strategy dimension).

Definition 2. *A **design dimension** is a tuple $dd=\langle VP, targetElementsExp\rangle$, where VP is a non-empty totally ordered set of variation points and targetElementsExp is an OCL (Object Constraint Language) expression which returns – when evaluated on an initial UML architectural model M – the elements of M that demand a decision about the architectural concern represented by dd. Such elements are named **target elements** of dd with respect to M and denoted by $targetElements(dd, M)$.*

The *targetElementsExp* expression relies on the associated UML profile's annotations to detect, in the initial model, the architectural *loci* that demand decisions about such concern. For instance, an initial model may require the choice of particular control strategies for two different service components. Therefore, two instances of the control strategy design dimension are created to capture the decisions for those architectural *loci*.

Definition 3. *A **design dimension instance** is a tuple $ddi=\langle M, dd, te\rangle$, where M is an initial UML architectural model, dd is a design dimension, and te is a target element of dd with respect to M.*

A variation point describes the elements (architectural extensions) that must be added to the initial model in order to implement such particular solution.

Definition 4. *A **variation point** is a tuple $vp=\langle C, postConditionExp\rangle$, where C is a totally ordered set of architectural changes and postConditionExp is an OCL expression evaluated after all changes in C are applied in the initial model. Such an expression must return true for valid architectures or false otherwise.*

Definition 5. *An **architectural change** c is a single indivisible operation that, when applied to a model M, results in a model $M' \neq M$. An architectural change c may represent an element addition, element removal or element's property change.*

The set of all design dimension instances generated by *ds*, when evaluated in *M*, provides the underlying infrastructure of our search-based approach for automating the architecture design process.

Definition 6. *An **application specific design space** is a tuple $asds=\langle M, ds, DDI\rangle$, where M is an initial UML architectural model, ds is a design space, and DDI is a partially ordered set of design dimension instances, defined as:*

$$DDI = \bigcup_{dd \in ds.DD} \left(\bigcup_{te \in dd.targetElements(M)} ddi = \langle M, dd, te \rangle \right) \tag{1}$$

The ultimate decision space may then be specified in terms of an application specific design space.

Definition 7. *The **architectural decision space** $\mathcal{D}_{asds}$ for a given application specific design space asds is the Cartesian product of all variation point indexes of design dimensions associated to each design dimension instance in asds.DDI:*

$$\begin{aligned} \mathcal{D}_{asds} = {} & \{1, 2, \ldots, |asds.DDI_1.dd.VP|\} \times \\ & \{1, 2, \ldots, |asds.DDI_2.dd.VP|\} \times \\ & \ldots \\ & \{1, 2, \ldots, \left|asds.DDI_{|DDI|}.dd.VP\right|\} \end{aligned} \tag{2}$$

*A vector $\mathbf{x} \in \mathcal{D}_{asds}$ is named **candidate vector**. The architectural model resulting from the valid application of all changes of variation points whose indexes are described in $\mathbf{x}$ is named **candidate architecture**. The subset of $\mathcal{D}_{asds}$ formed only by those candidate vectors resulting in valid architectures is named **architectural feasible space** ($\mathcal{F}_{asds}$).*

Therefore, a candidate architecture (a location in such n-dimensional space) is formed by the initial model extended with the merge of all architectural extensions provided by all involved variation points.

Finally, a quality metric may be defined for a given design space.

Definition 8. *A **quality metric** is a tuple $qm = \langle \Phi, g \rangle$. Φ is a function $\Phi \colon \mathcal{F}_{asds} \rightarrow \mathcal{V}$, where $\mathcal{F}_{asds}$ is an architectural feasible space and $\mathcal{V}$ is a set supporting measurements at least in interval scale (Stevens 1946). g must take the value 1 or -1 indicating, respectively, whether the metric should be maximized or minimized. The **architectural objective space** $\mathcal{O}_{asds}$ is defined as the Cartesian product $\mathcal{V}_1 \times \mathcal{V}_2 \times \ldots \times \mathcal{V}_n$, where $\mathcal{V}_i$ is the image of the function Φ_i (evaluation of the i-th metric of asds.ds.QM).*

As a consequence of such an infrastructure, huge design spaces may easily be spawned even for small input models, motivating the adoption of meta-heuristics and multi-objective optimization approaches. The number of different candidate vectors in $\mathcal{D}_{asds}$ (including those resulting in invalid architectures) is given by:

$$\prod_{dd \in ds.DD} |dd.VP|^{|dd.targetElements(M)|} \tag{3}$$

Once a concrete design space is defined, architects can submit initial models to manual design space exploration or rely on the multi-objective optimization engine we provide (*design space usage* stage). The domain-independent optimization engine we provide handles all required steps to forge candidate architectures for a given set of design space locations, evaluate their quality regarding the attributes defined for the design space, and find out a set of Pareto-optimal architectures.

Let $\Phi_{asds.ds.QM}(M_c)$ be the function that evaluate all quality metrics in *asds.ds.QM* with respect to a candidate architecture M_c:

$$\Phi_{QM}\colon \mathcal{F}_{asds} \rightarrow \mathcal{O} \tag{4}$$
$$\Phi_{QM}(M_r) \mapsto (-g_1 \cdot \Phi_1(M_r), -g_2 \cdot \Phi_2(M_r), \ldots, -g_n \cdot \Phi_n(M_r))$$

Let $\mathcal{T}\colon \mathbf{x}' \rightarrow M_c$ be the function that produces the candidate architecture M_c associated to a candidate vector $\mathbf{x}' \in \mathcal{F}_{asds}$. The optimization problem may then be stated as:

$$\overset{\prec}{\min}_{\mathbf{x}' \in \mathcal{F}_{asds}} \Phi_{QM}(\mathcal{T}(\mathbf{x}')) \tag{5}$$

where $\overset{\prec}{\min}$ denote minimization for Pareto optimality (Deb and Kalyanmoy 2001). Further information about the DuSE meta-model and its architecture optimization engine may be found in (Andrade and de Araújo Macêdo 2013a,b).

The aforementioned infrastructure provides the underpinnings of our SA systems design approach. We have specified a particular DuSE instance (SA:DuSE) that captures the most prominent degrees of freedom and quality attributes when designing adaptation loops based on feedback control (Tilbury et al. 2004). Currently, SA:DuSE yields architectural extensions regarding seven different control laws (Tilbury et al. 2004) (Proportional, Integral, Proportional-Integral, Proportional-Derivative, Proportional-Integral-Derivative, Static State Feedback, and Dynamic State Feedback), seven empirical tuning approaches (Wang 2005) (four Chien-Hrones-Reswick variations, Ziegler-Nichols, Cohen-Coon, and Linear Quadratic Regulator), five mechanisms for control adaptation (Landau et al. 2011) (fixed gain, gain scheduling, model-reference, model-identification, and reconfiguring control), and six different multiple loops arrangements (Weyns et al. 2010) (no cooperation, information sharing, coordinated control, regional planning, master/slave, and hierarchical).

In addition, four quality metrics (objective functions) evaluate the resulting architectures regarding the average settling time, average overshoot, control overhead, and control robustness. The first three metrics are intended to be minimized, while the last one is intended to be maximized. It is well-known from studies (Tilbury et al. 2004) in control theory field that settling time and average overshoot represent conflicting control goals (as presented in Figure 2b). The same has been observed for control overhead and control robustness metrics. One of our research goals was to investigate to which extent the proposed SA:DuSE design space captures such trade-offs when automating the design of SA systems architectures (as discussed below). Moreover, the architectural decision space produced by SA:DuSE exhibited 8,643,600 candidate vectors for an input model with two controllable ports. For models with four controllable ports, such number rapidly increases to 7.4711821e13, further motivating the need for effective search-based approaches. Further information about the SA:DuSE design dimensions, its corresponding variation points, and the adopted quality metrics may be found in (Andrade and de Araújo Macêdo 2013a).

Our approach has been fully implemented in a supporting tool named DuSE-MT (http://duse.sf.net), developed using the C++ programming language and the Qt cross-platform toolkit (http://www.qt.io). DuSE-MT is a meta-model agnostic tool we develop in order to support general software modeling activities and, in particular, the automated

design process we propose. The NSGA-II evolutionary algorithm (Deb et al. 2002) is currently used as optimization back-end, but other approaches can be easily adopted in the future thanks to the DuSE-MT's plugin-based architecture and the optimization engine's internals we have designed.

Figure 4 depicts the scatter plot matrix for the population of architectures resulting from one optimization run using a web server architecture (presented in Section 5.1) as initial model. The matrix's main diagonal presents – for each quality metric – the histograms of solutions in the Pareto-front and of dominated solutions. The remaining cells present the projection of the final population with respect to the quality metrics indicated at the cell's row and column. For instance, the scatter plot at the first column and second row depicts solutions using average settling time values in the abscissa and average overshoot values in the ordinate. Solutions in the Pareto-front

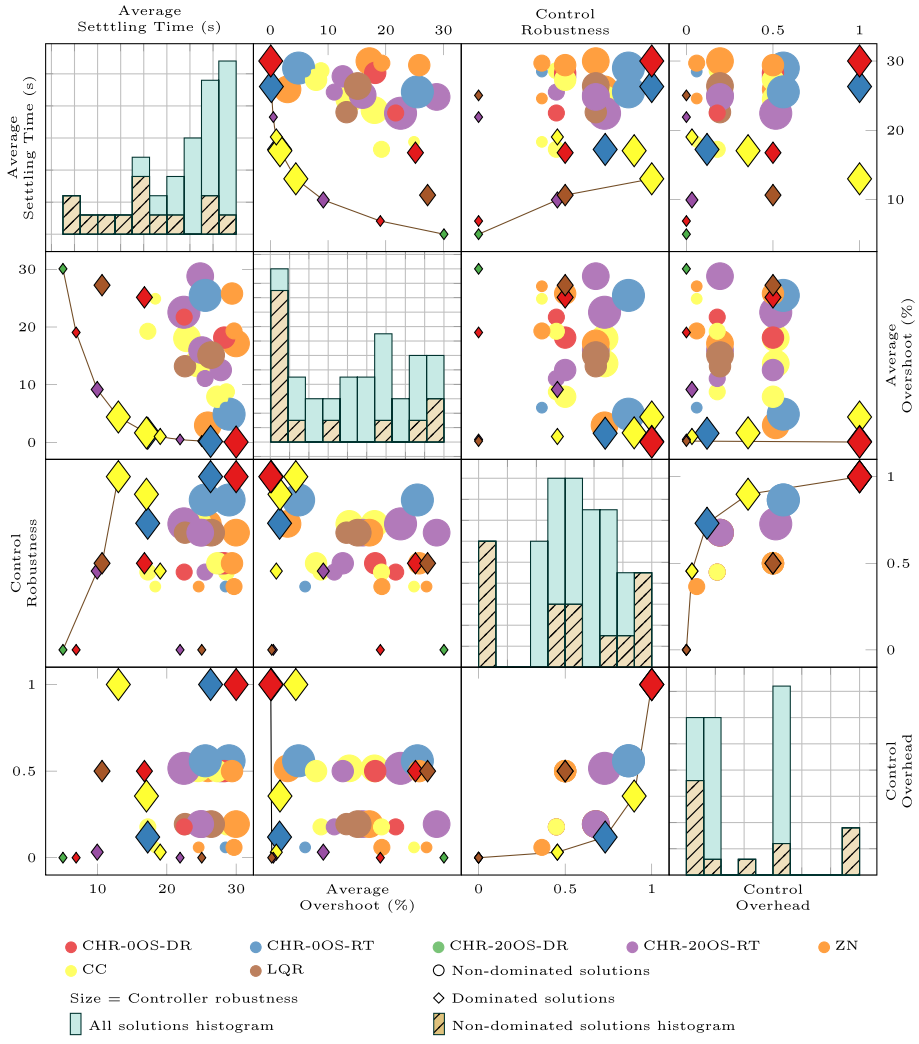

Figure 4 Scatter plot matrix depicting a population of architectures resulting from one optimization run using a web server initial model. The four charts at the matrix's main diagonal present the histograms of solutions in the Pareto-front and of dominated solutions. The remaining cells present the projection of the population for every pair of quality metrics.

are presented as diamonds, while the dominated ones are depicted as circles. A partial Pareto-front – regarding only the two quality metrics involved in a given cell – is shown as diamonds connected by a line. As indicated in the figure's legend, the solution color represents the controller tuning approach adopted by such an architecture. Finally, the solution size denotes the architecture's control robustness (the bigger, the more robust).

The outcome of our approach provides useful insights and supports the self-adaptive systems architect in several aspects. First, we observe that architectures exhibiting short average settling times are quite rare in the final population, making it harder for novice architects to find out such effective solutions by manually scouring the design space or by performing random searches. Second, the outcome reveals pronounced trade-offs between two pairs of quality attributes: *i)* average settling time and average overshoot (first column, second row); and *ii)* control robustness and control overhead (third column, fourth row). The Pareto-fronts for such combinations are smooth, providing alternative solutions regarding the fulfillment of such quality attributes. No significant trade-offs have been found in other quality metric pairs. Third, the rigorous identification of Pareto-optimal solutions prevents novice architects from adopting those combinations of control law, tuning technique, and control adaptation mechanism that lead to inferior architectures. Finally, the metric values presented by solutions in the Pareto-front allow for the early analysis of the dynamics exhibited by real prototypes implementing such architectures.

As part of the activities we have been conducting for evaluating our approach, in this work we look for any empirical evidence supporting the claim that search-based approaches improve the effectiveness and reduce the complexity of SA systems architectures. Furthermore, we want to know whether search-based approaches leverage the acquisition of distilled design knowledge by novice architects.

4 Methods

The goal of the experiment we report herein was to **analyze** the design of SA systems, **for the purpose of** evaluating the search-based design approach we propose and a design process based on architecture styles catalogs, **with respect to** the effectiveness and complexity of resulting architectures, as well as the method's potential for leveraging the acquisition of distilled design knowledge by novice SA systems architects, **from the viewpoint of** researchers, and **in the context of** graduate students endowing systems with self-adaptation capabilities.

The quasi-experiment (Wohlin et al. 2012) is characterized as a blocked subject-object study with a paired comparison design. Two UML models representing a web server and a MapReduce distributed architecture are used as experiment objects and two treatments (search-based approach and style-based approach) are considered for the design method factor (independent variable). We use the Generational Distance metric (Deb and Kalyanmoy 2001; Van Veldhuizen and Lamont 2000) to assess effectiveness in terms of how far the architectures designed by the subjects are from a previously determined Pareto-optimal set of architectures. Design complexity is evaluated by using the Component Point approach (Wijayasiriwardhane and Lai 2010) while a questionnaire with multiple choice questions evaluates the method's potential for leveraging the acquisition of design knowledge.

5 Experiment planning

The experiment took place as part of a 32 hours course on Software Engineering for Distributed Systems, arranged in eight classes (four hours each) along four weeks. As presented in Table 1, the course was split in three parts: lectures, exam and training, and experiment. In the first four classes, students were exposed to the foundations of SA systems and feedback control, as well as to the SISO (Single-Input Single-Output) and MIMO (Multiple-Input Multiple-Output) feedback control strategies (Tilbury et al. 2004) more widely adopted in SA systems. It is worth mentioning that all students had previously undertaken a 32 hours course on Software Architecture and Software Modeling.

Roughly half of the students work as software developers/designers, while the remaining have a stronger background in network administration. We try to insulate the effect of this factor by using blocking techniques as described in Section 5.4. Furthermore, no explicit guidance about self-adaptation quality attributes trade-offs was given during the lectures, since such an aspect is part of the hypotheses investigated herein. In the 5th day, we conducted an one hour discussion about the matter, followed up by a three hours exam where students used pen and paper to answer open-ended questions. In the 6th day, we discussed the exam results and presented a 3 hours training session about the DuSE-MT tool and the architecture styles catalog for SA systems we developed for this experiment.

The experiment took place in the last two days of the course. Students were randomly assigned to two equal size groups, blocked by their stronger technical background (see Section 5.4). Since we undertook the experiment as a blocked subject-object study with three objects (web server initial model, MapReduce architecture initial model, and questionnaire) and two treatments (search-based approach and style-based approach), a total of twelve tests were undertaken (presented in Table 2 and discussed later in Section 5.4). Each group experienced every combination of an experiment object and a treatment, exchanging the first experienced combination at the second day in order to minimize maturation threats. All design tests aimed at extending an initial model with a SA mechanism which regulates a performance metric, while yet minimizing the settling time, maximum

Table 1 Overview of the 32 h course in which the experiment was undertaken

Part	Day	Activity
Lectures	1	Self-adaptive systems foundations (motivation, MAPE-K reference architecture, current approaches, challenges)
	2	Feedback control introduction (control goals, control properties, fixed gain SISO approaches)
	3	Feedback control (MIMO and adaptive approaches)
	4	Self-adaptive systems - case studies
Exam and training	5	First hour: discussion Next 3 hours: Pen and paper exam
	6	First hour: exam discussion Next 3 hours: Training (DuSE-MT and architectural styles catalog)
Experiment	7	First 110 minutes: Tests #1 and #2 Next 110 minutes: Tests #3 and #4 Next 20 minutes: Tests #5 and #6 (questionnaire)
	8	First 110 minutes: Tests #7 and #8 Next 110 minutes: Tests #9 and #10 Next 20 minutes: Tests #11 and #12 (questionnaire)

Table 2 Tests defined for the experiment

#Test	Object	Treatment	Subjects
1	Web server	Style-based approach	Group 1
2	MapReduce architecture	Search-based approach	Group 2
3	MapReduce architecture	Style-based approach	Group 1
4	Web server	Search-based approach	Group 2
5	Questionnaire	Style-based approach	Group 1
6	Questionnaire	Search-based approach	Group 2
7	MapReduce architecture	Search-based approach	Group 1
8	Web server	Style-based approach	Group 2
9	Web server	Search-based approach	Group 1
10	MapReduce architecture	Style-based approach	Group 2
11	Questionnaire	Search-based approach	Group 1
12	Questionnaire	Style-based approach	Group 2

overshoot, and control overhead. Both groups used DuSE-MT as the design tool, but all functionalities regarding design space navigation and architecture optimization were turned off when using the style-based approach as treatment. Conversely, students had no access to the style catalog when using the search-based approach. We would like to emphasize that the experiment was not intended to investigate design productivity and fault density, since those aspects are obviously favored when adopting automated design approaches.

5.1 Design objects

The experiment's design tests aimed to create managing elements (adaptation loops) for two distinct managed elements: a web server and a MapReduce distributed architecture (Dean and Ghemawat 2008). Such managed elements were used as experiment objects and are depicted in Figure 5a and 5b as elements with the "*input model elements*" key. Experiment subjects were expected to extend such input models with a particular feedback loop design that produces short settling times, minimum overshoot, and low control overhead. Figure 5 shows two examples of such loops as elements with the "*added elements*" key. We chose these experiment objects because they constitute two self-adaptation scenarios widely investigated nowadays and pose different design challenges: MIMO local control for the web server case study vs. SISO nested control in a distributed environment for the MapReduce architecture case study.

The web server model (WS) – depicted in Figure 5a – entails a single component providing four interfaces: two for monitoring purposes (`IAvgCPUUtilization` and `IAvgMemUtilization`) and two for adjusting parameters that directly impacts the measured outputs (`IKeepAliveTimeout` and `IMaxRequestWorkers`). The goal is to retain web server's CPU and memory utilization as close as possible to the specified reference values, by simultaneously adjusting the number of threads serving HTTP requests (via `IMaxRequestWorkers` interface) and the amount of time the server must wait for subsequent requests on a given connection (via `IKeepAliveTimeout` interface).

The MapReduce architecture model (MR) – depicted in Figure 5b – describes a distributed computing infrastructure (cluster) where an array of n nodes stores and analyzes huge datasets. The cluster infrastructure orchestrates the parallel execution of a Map function for each data block stored in cluster's nodes and combines all

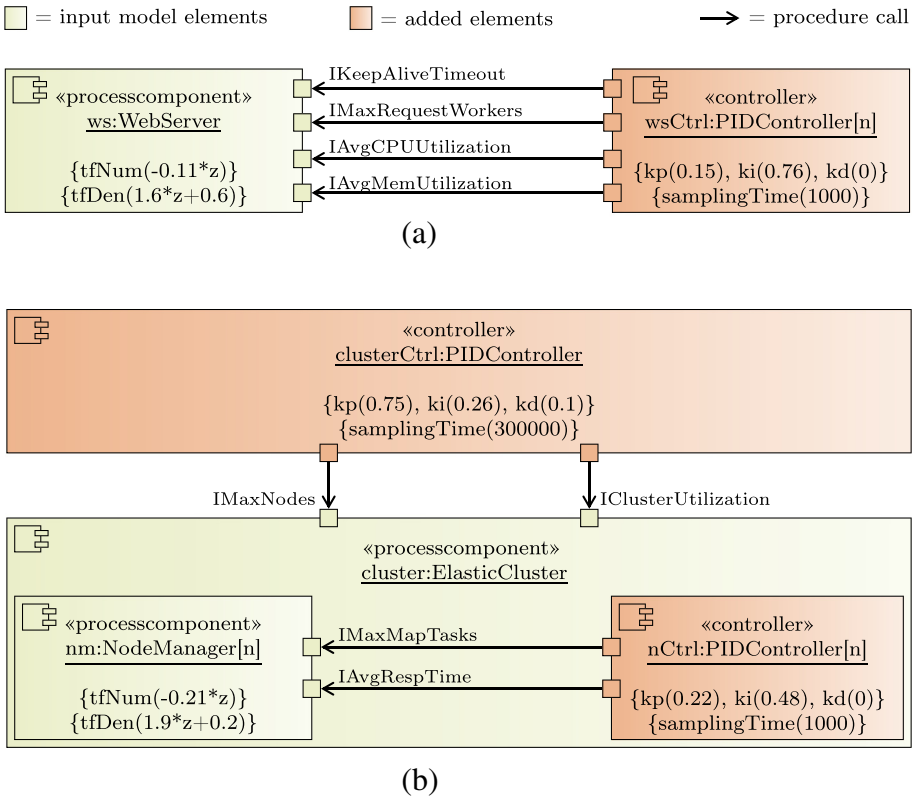

Figure 5 Experiment objects (input model elements only): web server model (a) and MapReduce architecture model (b). The added elements exemplify the architectural extensions to be designed by the experiment subjects, either by applying the architectural styles catalog (reference approach) or by adopting our automated design space exploration mechanism (intervention approach).

Map's outputs to form the Reduce function's input (Dean and Ghemawat 2008). Apache Hadoop (White 2009) is a well-established open source implementation of the MapReduce programming model, whose performance may be fine-tuned through nearly 190 configuration parameters. Although default values for such parameters are already provided by Hadoop, improvements of 50% in performance have been observed in properly configured setups (Jiang et al. 2010). In spite of that, Hadoop provides no services for parameter self-optimization or feedback control loops. The model we present in Figure 5b entails two nested controllable components: `NodeManager` and `ElasticCluster`. Each cluster machine runs the `NodeManager` service, which may have its partial job's average response time (measured via `IAvgRespTime` interface) regulated by adjusting the maximum number of map tasks simultaneously executing in that host (Hadoop's `mapreduce.tasktracker.map.tasks.maximum` parameter, changed via `IMaxMapTasks` interface). Additionally, the overall cluster utilization (measured via `IClusterUtilization` interface) may also be regulated by adjusting the number of cluster hosts serving the job (via `IMaxNodes` interface).

5.2 Variables selection

In this quasi-experiment, we are interested in analyzing the impact of the adopted design method on three dependent variables: the effectiveness of the resulting managing element, the complexity of managing element's architecture, and the method's

potential for promoting the acquisition of insights and refined experience about quality attributes trade-offs involved in SA systems design. This subsection describes the metrics we adopted to quantify such variables.

5.2.1 Measuring effectiveness (Generational distance)

We quantify the effectiveness of resulting feedback control loops in terms of how close their quality attributes are from a set of Pareto-optimal solutions previously obtained. Since meta-heuristics-based approaches – like ours – do not guarantee global optimality, we performed a set of 50 optimization runs and calculated the reference Pareto-front P^* of the union of all runs' outputs. A Pareto-front is a set of solutions for which it is impossible to make any other architecture better off without make at least another one worse. We assume that P^* (triangle path in Figure 6) is a nice representative of the most effective solutions and constitutes a reasonable reference value for evaluating how effective are the architectures designed by the experiment subjects. We have done such procedure for both the objects (WS and MR) used in the experiment, producing the P^*_{WS} and P^*_{MR} reference Pareto-fronts.

The Generational Distance (*GD*) (Deb and Kalyanmoy 2001; Van Veldhuizen and Lamont 2000) is a widely used metric to evaluate closeness between two Pareto-Fronts Q and P^*. The metric finds an average distance of the solutions of Q (or r) from P^*, as follows:

$$GD = \frac{\left(\sum_{i=1}^{|Q|} d_i^p\right)^{1/p}}{|Q|}; \quad \text{where } d_i^2 = \min_{k=1}^{|P*|} \sqrt{\sum_{m=1}^{M} \left(f_m^{(i)} - f*_m^{(k)}\right)^2} \tag{6}$$

$f_m^{(i)}$ is the m-th objective function value of the i-th member of Pareto-front Q. d_i^2 calculates the shortest distance between $f_m^{(i)}$ and $f*_m^{(k)}$: the m-th objective function value of the k-th member of Pareto-front P^*. Any L^p-norm can be used in Generational Distance. For $p = 2$ as described above, d_i^2 is the Euclidean distance between the solution $i \in Q$ and the nearest member of P^*. We chose the Generational Distance because it is more suitable

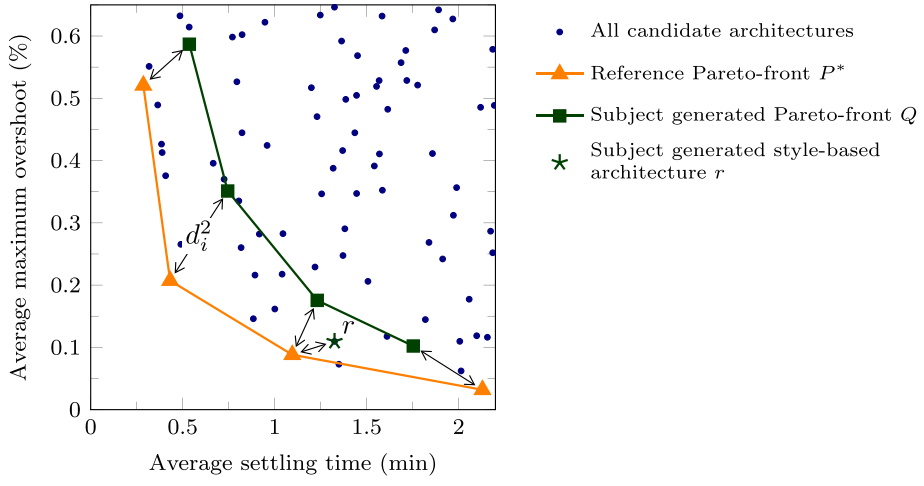

Figure 6 Example of reference Pareto-front *P and Generational Distance (*GD*) metric.** *GD* finds the average Euclidean distance from each *i*-th solution in *Q* to the nearest solution in *P**. The lower the *GD* value, the more effective are the architectures regarding self-adaptation quality attributes. The style-based architecture *r* is shown as more effective than those produced by the search-based approach. That is one of the null hypotheses investigated herein.

than alternatives like Error Ratio and Set Coverage (Deb and Kalyanmoy 2001) when comparing disjoint Pareto-fronts. Furthermore, its evaluation can be performed with lower computational costs when compared to metrics such as Hypervolume (van Veldhuizen and Lamont 1999) and Attainment Surface Based Statistical Metric (Fonseca and Fleming 1996).

As for designs produced with the style-based approach (e.g.: solution *r* in Figure 6), their corresponding location in objective space (quality metrics values) were first calculated and then compared to the reference Pareto-front using Generational Distance. Note that a Pareto-front Q obtained with the search-based approach (e.g.: square path in Figure 6) is not necessarily as effective as the reference Pareto-front P^*, because of the inherent randomness in the adopted evolutionary multi-objective optimization approach (NSGA-II).

Although r is shown, in Figure 6, more effective than any solution in Q, we believe that this is unlikely to happen in designs undertaken by architects with no previous experience in SA systems. Therefore, in this experiment, we look for any evidence that supports/rejects our claim that search-based approaches may improve the effectiveness of such designs.

5.2.2 Measuring complexity (Component Point)

The second dependent variable we focus in this experiment is design complexity, since it directly impacts the development effort required to implement the proposed architectures. We used the Component Point (CP) approach (Wijayasiriwardhane and Lai 2010) to quantify such an aspect, motivated by its original conception towards the measurement of UML models and by the existence of empirical evidence regarding its validity and usefulness (Wijayasiriwardhane and Lai 2010). CP provides the means to measure design complexity in terms of component's interfaces complexity and component's interaction complexity. We define the complexity CC_c for a component c as:

$$CC_c = IFCI_c + ITCI_c = \frac{IFC_c}{n_c} + \frac{ITC_c}{m_c} \tag{7}$$

$IFCI_c$ is the Interface Complexity per Interface, defined as the component's Interface Complexity (IFC_c) divided by the number of component's provided interfaces (n_c). Similarly, the Interaction Complexity per Interaction ($ITCI_c$) is defined as the component's Interaction Complexity (ITC_c) divided by the number of component's interactions (m_c). IFC_c and ITC_c, in their turn, are defined as follows.

The first step when calculating IFC_c is classifying each interface of a component into two types: ILF (Internal Logical Files) or EIF (External Interface Files). ILF interfaces are those whose operations change attributes of other interfaces, while the remaining interfaces as classified as EIF. The CP approach also specifies how a complexity level (Low, Average, High) should be assigned to each interface, based on the number of operations and number of operation's parameters it presents. Hence, IFC_c is defined as:

$$IFC_c = \sum_{j=1}^{2} \sum_{k=1}^{3} I_{jk} \times W_{jk} \tag{8}$$

I_{jk} is the number of interfaces of type j (1=ILF and 2=EIF) with complexity level k (1=Low, 2=Average, and 3=High). W_{jk} is the weight, given by the CP approach, for the interface type j with complexity level k.

ITC_c is evaluated in terms of the Interaction Frequency (IF_{ij}) of the j-th operation of the i-th interface and the Complexity Measure (CM_{ijk}) of the k-th data type involved in the execution of the j-th operation of the i-th interface. IF_{ij} is defined as a ratio of the number of interactions (N_O) performed by the operation and the number of interactions (N_I) performed by all operations of the interface. CM_{ijk}, in its turn, is defined as:

$$CM_{ijk}(D, L) = L + \sum_{n=1}^{m} CM(DT_n, L + 1) \tag{9}$$

D is the data type under measurement, L is the number of the level where the data type D occurs in the component data type graph (initially 1), DT_n is the data type of the n-th D's data member and m is the number of data members in D. Finally, ITC_c can be defined as:

$$ITC_c = \sum_{i=1}^{p} \sum_{j=1}^{q} \left(IF_{ij} \times \sum_{k=1}^{r} CM_{ijk} \right) \tag{10}$$

p is the number of interfaces provided by component c, q is the number of operations that the i-th interface provides, and r is the number of data types involved in the execution of the j-th operation of the i-th interface. The overall architecture complexity AC is defined as the sum of the CC_i of every component i comprising the solution.

5.2.3 Measuring the acquisition of distilled design knowledge (post-experiment questionnaire)

The third dependent variable we investigated herein is the method's potential for leveraging the acquisition of distilled design knowledge. For that purpose, we prepared a questionnaire with 10 multiple choice questions related to refined knowledge about quality attribute trade-offs in the SA systems domain. Students answered such questionnaire at the end of each experiment day and we assigned grades according to the number of correctly answered questions. The goal was to evaluate to which extent the adopted design approach may leverage the acquisition of distilled knowledge about such design trade-offs. The questionnaire is available at the experiment website.

5.3 Hypotheses formulation

In the quasi-experiment we report herein, we compare the use of a search-based architecture design approach and a style-based design approach with respect to the effectiveness and complexity of resulting architectures, as well as to the method's potential to promote the acquisition of distilled design knowledge. Such goal has been stated in three null hypotheses (H_0) and their corresponding alternative hypotheses (H_1):

- H_0^1: there is no difference in *design effectiveness* (measured in terms of the Generational Distance GD) between a feedback control loop design created using the style-based approach (reference approach: RA) and a feedback control loop design created using the search-based approach (intervention approach: IA).

$$H_0^1 : \mu_{GD_{RA}} = \mu_{GD_{IA}}$$
$$H_1^1 : \mu_{GD_{RA}} > \mu_{GD_{IA}}$$

- H_0^2: there is no difference in *design complexity* (measured in terms of the Architectural Complexity AC) between a feedback control loop design created using

the style-based approach and a feedback control loop design created using the search-based approach.

$$H_0^2 : \mu_{AC_{RA}} = \mu_{AC_{IA}}$$
$$H_1^2 : \mu_{AC_{RA}} > \mu_{AC_{IA}}$$

- H_0^3: there is no difference in the *acquisition of distilled design knowledge* (measured in terms of applied questionnaire's grade QG) between designing a feedback control loop using the style-based approach and designing a feedback control loop using the search-based approach.

$$H_0^3 : \mu_{QG_{RA}} = \mu_{QG_{IA}}$$
$$H_1^3 : \mu_{QG_{RA}} < \mu_{QG_{IA}}$$

5.4 Experiment design

The experiment was undertaken as a blocked subject-object study, which means that each subject exercises both treatments and that effects can be compared in pairs. Since the experiment students had a stronger technical background in two different fields (14 devoted to software development and 10 devoted to network administration), we used such an information as a blocking factor. By doing that, we eliminate the undesired effect of student's technical background on the dependent variables, increasing the precision of the experiment.

Students from each technical background partition were randomly and equally assigned to groups 1 and 2, yielding a similar proportion of developers and network administrators in each group. As presented in Table 2, a total of eight design tests and four questionnaire answering tests were conducted in the experiment. Each group experienced every combination of an object (WS model, MR model, or the questionnaire) and a treatment (style-based approach or search-based approach). In the first experiment day, group 1 applied the style-based approach, initially in the WS model and then in the MR model, while group 2 adopted the search-based approach with the opposite object order. At the end of the day, both groups answered the questionnaire based on their experience with the corresponding approach. In the second experiment day, groups exchanged the treatments and experienced the objects in the opposite order to the one conducted by them in the previous day. The same questionnaire was applied again at the end of the second day.

To minimize the effect of subjects gaining information from previous assignments, we systematically balanced which object-treatment combination is first experienced in each group. The tests' operation order is presented in Table 1. To reduce hypotheses guessing and other social threats, students did not receive any feedback and were not aware of the experiment until its completion.

The architecture styles catalog we developed for this experiment documents the same design knowledge present in the SA:DuSE design space as a group of eleven architectural styles. Table 3 presents one of such styles. We describe each solution using a schema that documents the style's prominent components, connectors, and data elements, the resulting architecture topology, induced qualities, typical uses, and potential cautions, among other aspects.

Table 3 One of the eleven architectural styles for self-adaptive systems described in the catalog

Style #2 – P(ID) Feedback control with system identification	
Summary	A separate component (controller) measures system output and acts accordingly to drive the system to the expected output (feedback)
Components	System, controller, effector, sensor, transducer/QoS subsystem (optional, if output not directly delivered by the system itself)
Connectors	(Remote) procedure call, event bus or data access
Data elements	Reference value(s), input values, output values, transduced values (optional)
Topology	Circular with one entry point: (reference input → controller → system → measured output [→ transducer/QoS subsystem] → ...)
Variants	#2.1: PID-SI with Precompensation - (PID-SI/PC) #2.2: PID-SI with Sensor Delay - (PID-SI/SD) #2.3: PID-SI with Filtering - (PID-SI/F)
Qualities yielded	Reactive behavior; adaptation to unmodeled disturbances; no need for an accurate system model; can make stable an unstable system
Typical uses	When a good enough system model is available, disturbance modeling is quite complicated, target system is unstable but linear or with identifiable linear operating regions
Cautions	When disturbance spans over a wide range; when system is primary non-linear or have dynamics that are difficult to be modeled; when structural reconfiguration is needed
Example	...

As for the operation stage, we commit the participants by presenting some real cases demanding self-adaptation capabilities and explaining how the myriad of available approaches makes things harder for novice architects. Grades have been assigned to all tests as a form of inducement. Some instrumentation was required in order to enable/disable the search features in DuSE-MT and collect the resulting architectures from each participant. After the experiment operation, 20 subjects provided usable data for paired comparison of Generational Distance and Architecture Complexity. Questionnaire answers were restrict to those provided by such 20 subjects.

6 Results and discussion

With the support of DuSE-MT, all UML models resulting from the design tests were serialized in XML (eXtensible Markup Language) files, along with their corresponding quality attributes values (objective-space location). Such values were used to compute the Generational Distance for all resulting models. The Architecture Complexity value was also calculated for each resulting UML model.

6.1 Analysis

Table 4 and Figure 7 summarize the measured values of all dependent variables, as well as their paired difference with respect to the adopted treatment. The first step we undertook in the analysis stage was to investigate whether the usual assumptions for the use of parametric tests – preferable because of their enhanced power – hold in the collected data. Such assumptions are: *i)* data is taken from an interval or ratio scale (held for all experiment's dependent variables); *ii)* observations are independent (enforced by experiment design); *iii)* measured values are normally distributed in the populations; and *iv)* population variances are equal between groups (homoscedasticity).

Table 4 Descriptive statistics for the experiment's dependent variables

Generational distance (GD)			
	Mean(μ)	**Median**	**Std. dev.**
Search-based approach (IA)	2.40	2.45	1.08
Style-based approach (RA)	2.59	2.41	1.03
Difference (IA−RA)	-0.19	-0.62	1.32
Architecture complexity (AC)			
	Mean(μ)	**Median**	**Std. dev.**
Search-based approach (IA)	6.46	6.65	2.77
Style-based approach (RA)	7.02	7.05	2.70
Difference (IA−RA)	-0.57	-1.90	3.47
Questionnaire grade (QG)			
	Mean(μ)	**Median**	**Std. dev.**
Search-based approach (IA)	6.85	7.00	1.43
Style-based approach (RA)	7.04	7.25	1.27
Difference (IA−RA)	-0.19	-0.50	1.51

We used the Anderson-Darling test (Corder and Foreman 2009) to evaluate to which extent the paired differences are normally distributed. The Brown-Forsythe test (Good 2005) was applied to investigate the null hypothesis of homoscedasticity between the intervention approach and reference approach groups. Table 5 presents such a results. With a significance level (α) of 0.05, we observed that only the Questionnaire Grade (QG) paired difference could be considered normally distributed (Anderson-Darling p-value > 0.05). In addition, for all dependent variables, the null hypothesis of homoscedasticity could not be rejected (Brown-Forsythe p-value > 0.05). Since all assumptions must hold, only hypothesis H_0^3 was evaluated by using a parametric test. The paired differences of Generational Distance (GD) and Architecture Complexity (AC) were not considered normally distributed (p-value $< \alpha = 0.05$) and, as such, hypotheses H_0^1 and H_0^2 were evaluated by using a non-parametric test. As presented in Table 6, we used the Wilcoxon Signed-Rank test (Gibbons and Chakraborti 2003; Wohlin et al. 2012) to investigate H_0^1 and H_0^2 and the Paired t-test (Wohlin et al. 2012) to investigate H_0^3. With a significance level (α) of 0.05, H_0^1 and H_0^2 were rejected while no evidence could be found about H_0^3.

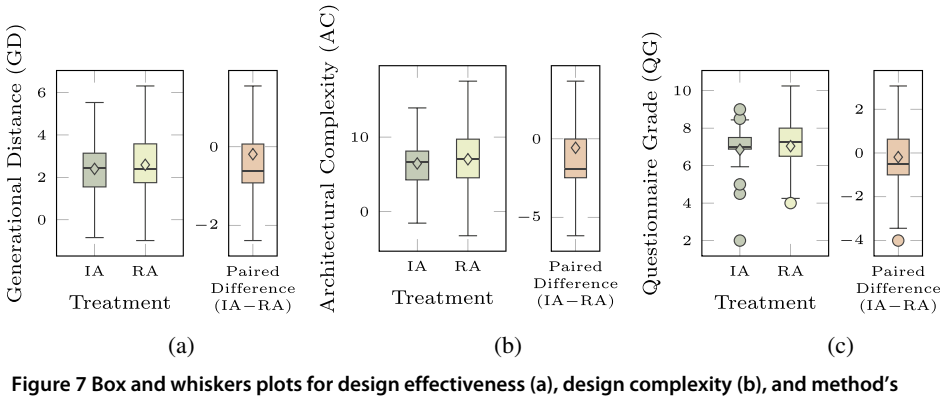

Figure 7 Box and whiskers plots for design effectiveness (a), design complexity (b), and method's potential for leveraging the acquisition of distilled knowledge (c). For each dependent variable it is shown the values for the search-based approach (IA: intervention approach), the style-based approach (RA: reference approach), as well as the values of the paired difference (IA−RA).

Table 5 Results of Anderson-Darling normality test and Brown-Forsythe heteroscedasticity test ($\alpha = 0.05$)

Dependent variable	Anderson-Darling *p*-value	Brown-Forsythe *p*-value
Generational distance	1.29814505086953E-007	0.9009324909
Architecture complexity	2.88672812219819E-010	0.7207666486
Questionnaire grade	0.635529605	0.7167840476

6.2 Discussion

The descriptive statistics and results of hypotheses tests show that there are improvements in the dependent variables, except for the Questionnaire Grade (QG). Actually, students who first exercised the search-based approach got slightly smaller grades (6.85) than other ones (7.04). Since both the architecture style catalog and the design space used in the experiment contain the same information, two possible reasons for such difference remain. First, students exposed to the search-based approach may have experienced a larger set of candidate architectures, which would contribute to obfuscate some quality trade-offs evaluated in the questionnaire. Second, the quality trade-offs may actually be not too difficult to grasp without the use of structured design spaces and automated architecture optimization, so that the difference is actually by chance. Further experiments are needed to better investigate such an aspect.

Generational Distance (adopted measure for effectiveness) is, on average, 7% lower with the search-based approach (2.40) when compared to the style-based approach (2.59). Architecture Complexity is, on average, 7% lower with the search-based approach (6.46) when compared to the style-based approach (7.02). While such values already indicate some improvements in the resulting architectures, we still lack further investigation about the boundaries that such enhancements may present.

7 Threats to validity

This section presents the threats to validity (Wohlin et al. 2012) we identified for the experiment.

7.1 Threats to construct validity

Construct Validity is the degree to which the objects and measurements reflect their associated constructs in the real world. We have identified three such threats: inadequate pre-operational explication of constructs, hypothesis guessing, and objects representativeness.

First, since the theory behind feedback control loops encompasses areas such as systems and signals, modeling of dynamic behavior, and analysis in frequency domain, students may have had no enough time to get a firm grasp about such mathematical background. To reduce this threat, we focused on requiring minimum knowledge about such as aspect and tried to leverage tool support regarding this issue in DuSE-MT. Second, students

Table 6 Results of statistical tests ($\alpha = 0.05$)

H_0^i	Test	Criteria	Conclusion
1	Wilcoxon signed-rank	T(410) > T-critical(378)	Rejected
2	Wilcoxon signed-rank	T(367) > T-critical(361)	Rejected
3	Paired *t*-test	*p*-value=0.5488018266	Not rejected

may have tried to perform better when using the search-based approach because it is the treatment proposed by the course holders. Moreover, authors involvement with the intervening approach may have lead to better training on the use of the search-based mechanism. To mitigate this issue, students were not aware of the experiment and were graded on all tests. Third, the objects used in the experiment may not actually reflect the kind of problems routinely faced in the SA systems domain. Since the two adopted self-adaptation scenarios have been repeatedly investigated in a number of recent papers, we believe they constitute interesting and representative examples of current practice.

7.2 Threats to internal validity

Internal Validity concerns in analyzing to which extent unknown factors may affect the dependent variables with respect to causality. We have identified two such threats. The first one is maturation, where subject gain insights from previous experiment sessions. To reduce this threat, we alternately assigned such objects during the two experiment days. The second is related to instrumentation. Since a new modeling tool was adopted in the experiment (DuSE-MT), that may have impacted in some extent the student's abilities for developing the required models.

7.3 Threats to external validity

External Validity is related to the ability of generalizing the experiment results to other settings. Since we used students of a graduate program in Distributed and Ubiquitous Computing, they may not represent the expected background in current industry practice.

7.4 Threats to conclusion validity

Conclusion Validity is related to the ability of generalizing the results to the overall concept or theory which supports the experiment. Since the experiment objects were created by us, there is a potential threat that such objects do not actually represent the problem under investigation. Such threat could have been reduced by relying on external SA systems experts to design such objects.

8 Related work

To the best of our knowledge, no controlled experiments regarding the use of search-based approaches when designing SA systems have been undertaken so far. However, we identified one experiment regarding SA systems design and a number of papers reporting on controlled experiments about software architecture design.

In (Weyns et al. 2013), the authors report the results of a quasi-experiment that investigates whether the use of external feedback loops (when compared with internal adaptation mechanisms) improves the design of SA systems. The design was evaluated with respect to design complexity (in terms of activity complexity and control flow complexity), fault density, and design productivity. The experiment shows that external feedback loops reduce the number of adopted control flow primitives, increasing the design's understandability and maintainability. They also observed improvements in design productivity when using external feedback loops, but found no significant effects on design complexity in terms of activity complexity. The experiment we present herein tackles a similar design issue but with different treatments, objects, measurements, and

hypotheses. While their experiment reveals evidence about the decoupling and reusability benefits of external feedback loops, we believe our experiment contributes by revealing the potential benefits of systematic design knowledge representation and search-based automated design approaches in such a domain.

A controlled experiment aimed at evaluating the impact of design rationale documentation techniques on effectiveness and efficiency of decision-making in the presence of requirements changing is presented in (Falessi et al. 2006). The results show that the use of such documentation technique significantly improves effectiveness but with no impacts on efficiency. In (Bratthall et al. 2000), the authors present a controlled experiment which evaluates the impact of the use of design rationale documentation on software evolution. They conclude that there are improvements in correctness and productivity when such documentation is available. Our search-based design approach – under evaluation herein – supports rationale documentation in terms of domain-specific design spaces. The experiment we report in this paper is ultimately assessing the impact of having such rationale documented in a structured and systematic way, in contrast to ad-hoc styles catalogs or unstructured rationale documents.

In (Golden et al. 2005), a controlled experiment was performed to evaluate the usefulness of architectural patterns when evolving architectures to support specific usability concerns. The authors conclude that usability concerns are amenable to be handled in architectural level and that architectural patterns can significantly leverage such an aspect. In the SA systems domain, architecture-centric approaches with explicit (first-class) representation of feedback loops have been advocated as a promising research direction (Brun et al. 2009; Hebig et al. 2010; Müller et al. 2008), due to their generality and support for early reasoning of self-adaptation quality attributes. The experiment we describe in this paper evaluates how search-based design approaches impact such first-class representation of feedback loops.

A controlled experiment about the impact of the use of design patterns on the productivity and correctness of software evolution activities is described in (Vokác et al. 2004). They conclude that each design pattern presents a specific impact on such dependent variables and, therefore, claim that design patterns should not be characterized as useful or harmful in general. In contrast, our experiment compares the use of two distinct representations of such distilled design knowledge: architectural styles vs. structured design spaces. Furthermore, we are interested in evaluating whether search-based design automation improves the effectiveness of SA systems.

With respect to software engineering mechanisms for SA systems, (Weyns et al. 2012) present FORMS (FOrmal Reference Model for Self-adaptation): an unifying reference model for formal specification of distributed SA systems. Their approach provides a small number of modeling elements capturing key design concerns in the SA systems domain. In contrast to our approach, FORMS provides no means for automated design of feedback loops and a steep learning curve may be experienced because of its rigorous formal underpinnings.

(Vogel and Giese 2012) propose a new modeling language for explicitly describing feedback control loops as runtime megamodels (multiple models@runtime). In contrast, our approach builds on top of widely accepted standards for modeling languages, like MOF (Meta Object Facility) and UML. Although our approach has been used as an offline design mechanism, future work includes moving such an infrastructure to runtime,

providing a models@runtime approach for Dynamic Adaptive Search-Based Software Engineering (Harman et al. 2012).

A UML profile for modeling feedback control loops as first-class entities is presented in (Hebig et al. 2010). In our mechanism, we go a step further towards the use of UML profiles as the underlying mechanism for identifying *loci* of architectural decisions, enabling automated design, and detecting invalid candidate architectures. (Cheng et al. 2006) present an adaptation language which relies on utility theory for handling self-adaptation in the presence of multiple objectives. *A priori* preference articulation methods – like utility functions – convert a multi-objective optimization problem into a single-objective one, but its effectiveness highly depends on an well-chosen preference vector. Our approach, on the other hand, accommodates the multi-objective nature of SA systems design as an essential aspect by using *a posteriori* preference articulation.

An on-line learning-based approach for handling unanticipated changes at runtime is presented in (Esfahani et al. 2013). Whilst we have considered in this work only feedback control as the enabling mechanism for self-adaptation, other strategies may be modeled as new variation points. (Křikava et al. 2012) propose a models@runtime approach which represents adaptation logic as networks of messaging passing actors. Our work, in contrast, leverages design reuse by requiring the use of highly distilled design knowledge only once – when designing a domain-specific DuSE design space. Thereafter, novice architects have better support for designing effective architectures and getting insights from the search activities.

9 Conclusions

This paper presented a quasi-experiment aimed at evaluating whether search-based architecture design approaches improve the effectiveness and complexity of SA systems when compared to style-based design approaches. To the best of our knowledge, this is the first endeavor in evaluating how search-based automated design impacts the quality of SA systems. The results reveal that the use of systematically structured design spaces and architecture optimization mechanisms indeed provide enhanced support to the evaluation of quality trade-offs, for the experiment objects considered herein.

Some insights have been identified from the experiment results. We found no evidence that search-based approaches leverage the acquisition of distilled design knowledge in the SA systems domain. Alternative instruments for evaluating such an aspect may be adopted in future research, enabling the eliciting of more elucidative conclusions. However, search-based design approaches do contribute in revealing architectures which indeed exhibit a near-optimal trade-off between quality attributes. In contrast, architects using the style-based approach are more likely to design sub-optimal architectures. Improved effectiveness results in managing elements with lower overhead and enhanced use of resources, leveraging the overall SA behavior. Moreover, designs with lower complexity were also obtained when using the search-based approach, fostered by the systematic representation of the architecture changes required to implement the involved feedback loops. As a consequence, one should expect positive effects in understandability, maintainability, and testability of development artifacts implementing such architectures.

A lot of current research are driving their efforts towards the establishment of principled and well-founded underpinnings for engineering software-intensive systems, specially in particular application domains like SA systems. The organization of software

design knowledge for routine use is mandatory if we are to realize the upcoming generation of software-intensive systems.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
Both authors planned the experiment. SSA carried it out, analyzed its results, and prepared the initial draft of the manuscript. Corrections and reviews were made by RJAM. Both authors read and approved the final manuscript.

Received: 1 December 2014 Accepted: 3 March 2015
Published online: 24 March 2015

References

Andersson J, Lemos R, Malek S, Weyns D (2009) Software engineering for self-adaptive systems. In: Cheng BH, Lemos R, Giese H, Inverardi P, Magee J (eds). Software Engineering for Self-Adaptive Systems. Springer, Berlin, Heidelberg. pp 27–47. Chap. Modeling Dimensions of Self-Adaptive Software Systems doi:10.1007/978-3-642-02161-9_2

Andrade SS, de Araújo Macêdo RJ (2013) A search-based approach for architectural design of feedback control concerns in self-adaptive systems. In: 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2013, Philadelphia, PA, USA, September 9-13, 2013. IEEE Computer Society, Washington, DC, USA. pp 61–70. doi:10.1109/SASO.2013.42. http://dx.doi.org/10.1109/SASO.2013.42

Andrade SS, de Araújo Macêdo RJ (2013) Architectural design spaces for feedback control concerns in self-adaptive systems (S). In: The 25th International Conference on Software Engineering and Knowledge Engineering, Boston, MA, USA, June 27-29, 2013. Knowledge Systems Institute Graduate School, Skokie, Illinois, USA. pp 741–746

Andrade SS, de Araújo Macêdom RJ (2014) Do search-based approaches improve the design of self-adaptive systems? A controlled experiment. In: 2014 Brazilian Symposium on Software Engineering, Maceió, Brazil, September 28 - October 3, 2014. IEEE, Washington, DC, USA. pp 101–110. doi:10.1109/SBES.2014.17. http://dx.doi.org/10.1109/SBES.2014.17

DARPA (1997) Self-adaptive software. Technical Report 98-12, Defense Advanced Research Projects Agency

Bratthall L, Johansson E, Regnell B (2000) Is a design rationale vital when predicting change impact? A controlled experiment on software architecture evolution. In: Bomarius F, Oivo M (eds). Product Focused Software Process Improvement, Second International Conference, PROFES 2000, Oulu, Finland, June 20-22, 2000, Proceedings. Lecture Notes in Computer Science. Springer, New York, NY, USA Vol. 1840. pp 126–139. doi:10.1007/978-3-540-45051-1_14. http://dx.doi.org/10.1007/978-3-540-45051-1_14

Benyon D, Murray D (1993) Adaptive systems: from intelligent tutoring to autonomous agents. Knowledge Based Systems 6(4):179–219. doi:10.1016/0950-7051(93)90012-I

Bruni R, Bucchiarone A, Gnesi S, Melgratti H (2008) Modelling dynamic software architectures using typed graph grammars. Electronic Notes in Theoretical Computer Science 213(1):39–53. doi:10.1016/j.entcs.2008.04.073. Proceedings of the Third Workshop on Graph Transformation for Concurrency and Verification (GT-VC 2007)

Brun Y, Serugendo GDM, Gacek C, Giese H, Kienle HM, Litoiu M, Müller HA, Pezzè M, Shaw M (2009) Engineering self-adaptive systems through feedback loops. In: Cheng BHC, de Lemos R, Giese H, Inverardi P, Magee J (eds). Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]. Lecture Notes in Computer Science. Springer, New York, NY, USA Vol. 5525. pp 48–70. doi:10.1007/978-3-642-02161-9_3. http://dx.doi.org/10.1007/978-3-642-02161-9_3

Cheng S-W, Garlan D, Schmerl B (2006) Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems. SEAMS 2006. ACM, New York, NY, USA. pp 2–8. doi:10.1145/1137677.1137679. http://doi.acm.org/10.1145/1137677.1137679

Corder GW, Foreman DI (2009) Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach. Wiley, Hoboken, NJ, USA

de Lemos R, Giese H, Müller HA, Shaw M, Andersson J, Litoiu M, Schmerl BR, Tamura G, Villegas NM, Vogel T, Weyns D, Baresi L, Becker B, Bencomo N, Brun Y, Cukic B, Desmarais R, Dustdar S, Engels G, Geihs K, Göschka KM, Gorla A, Grassi V, Inverardi P, Karsai G, Kramer J, Lopes A, Magee J, Malek S, Mankovski S, *et al* (2010) Software engineering for self-adaptive systems: A second research roadmap. In: de Lemos R, Giese H, Müller HA, Shaw M (eds). Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers. Lecture Notes in Computer Science. Springer, New York, NY, USA Vol. 7475. pp 1–32. doi:10.1007/978-3-642-35813-5_1. http://dx.doi.org/10.1007/978-3-642-35813-5_1

Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Communications of the ACM 51(1):107–113. doi:10.1145/1327452.1327492

Deb K, Kalyanmoy D (2001) Multi-Objective Optimization Using Evolutionary Algorithms. John Wiley & Sons, Inc., New York, NY, USA

Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6(2):182–197. doi:10.1109/4235.996017

Esfahani N, Elkhodary A, Malek S (2013) A learning-based framework for engineering feature-oriented self-adaptive software systems. IEEE Transactions on Software Engineering 39(11):1467–1493. doi:10.1109/TSE.2013.37

Falessi D, Cantone G, Becker M (2006) Documenting design decision rationale to improve individual and team design decision making: an experimental evaluation. In: Travassos GH, Maldonado JC, Wohlin C (eds). 2006 International Symposium on Empirical Software Engineering (ISESE 2006), September 21-22, 2006, Rio de Janeiro, Brazil. ACM, New York, NY, USA. pp 134–143. doi:10.1145/1159733.1159755. http://doi.acm.org/10.1145/1159733.1159755

Fonseca CM, Fleming PJ (1996) On the performance assessment and comparison of stochastic multiobjective optimizers. In: Voigt H, Ebeling W, Rechenberger I, Schwefel H (eds). Parallel Problem Solving from Nature - PPSN IV, International

Conference on Evolutionary Computation. The 4th International Conference on Parallel Problem Solving from Nature, Berlin, Germany, September 22-26, 1996, Proceedings. Lecture Notes in Computer Science. Springer, New York, NY, USA Vol. 1141. pp 584–593. doi:10.1007/3-540-61723-X_1022. http://dx.doi.org/10.1007/3-540-61723-X_1022

Georgas JC, Taylor RN (2008) Policy-based self-adaptive architectures: a feasibility study in the robotics domain. In: Cheng BHC, de Lemos R, Garlan D, Giese H, Litoiu M, Magee J, Müller HA, Taylor RN (eds). 2008 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2008, Leipzig, Germany, May 12-13, 2008. ACM, New York, NY, USA. pp 105–112. doi:10.1145/1370018.1370038. http://doi.acm.org/10.1145/1370018.1370038

Georgiadis I, Magee J, Kramer J (2002) Self-organising software architectures for distributed systems. In: Garlan D, Kramer J, Wolf AL (eds). Proceedings of the First Workshop on Self-Healing Systems, WOSS 2002, Charleston, South Carolina, USA, November 18-19, 2002. ACM, New York, NY, USA. pp 33–38. doi:10.1145/582128.582135

Gibbons JD, Chakraborti S (2003) Nonparametric Statistical Inference, Fourth Edition: Revised and Expanded. Statistics: A Series of Textbooks and Monographs. Taylor & Francis, Florence, Kentucky, USA

Golden E, John BE, Bass L (2005) The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment. In: Roman G, Griswold WG, Nuseibeh B (eds). 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA. ACM, New York, NY, USA. pp 460–469. doi:10.1145/1062455.1062538. http://doi.acm.org/10.1145/1062455.1062538

Good PI (2005) Permutation, Parametric and Bootstrap Tests of Hypotheses, Vol. 3. Springer, New York, NY, USA

Harman M, Mansouri SA, Zhang Y (2012) Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys 45(1):11–11161. doi:10.1145/2379776.2379787

Harman M (2010) Why the virtual nature of software makes it ideal for search based optimization. In: Rosenblum DS, Taentzer G (eds). Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science. Springer, New York, NY, USA Vol. 6013. pp 1–12. doi:10.1007/978-3-642-12029-9_1. http://dx.doi.org/10.1007/978-3-642-12029-9_1

Harman M, Burke EK, Clark JA, Yao X (2012) Dynamic adaptive search based software engineering. In: Runeson P, Höst M, Mendes E, Andrews AA, Harrison R (eds). ACM-IEEE international symposium on Empirical software engineering and measurement. ACM, New York, NY, USA. pp 1–8

Hayes M (2011) Schaums Outline of Digital Signal Processing, 2nd Edition, Schaum's Outline Series. McGraw-Hill Education, New York, NY, USA

Hebig R, Giese H, Becker B (2010) Making control loops explicit when architecting self-adaptive systems. In: SOAR 2010: Proceedings of the Second International Workshop on Self-Organizing Architectures. ACM, Washington, DC, USA. pp 21–28

Huebscher MC, McCann JA (2008) A survey of autonomic computing – degrees, models, and applications. ACM Computing Surveys 40(3):7–1728. doi:10.1145/1380584.1380585

Jiang D, Ooi BC, Shi L, Wu S (2010) The performance of mapreduce: An in-depth study. Proceedings of the VLDB Endowment 3(1):472–483

Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50. doi:10.1109/MC.2003.1160055

Křikava F, Collet P, France RB (2012) Actor-based runtime model of adaptable feedback control loops. In: Proceedings of the 7th Workshop on Models@run.time, MRT 2012. ACM, New York, NY, USA. pp 39–44. doi:10.1145/2422518.2422525. http://doi.acm.org/10.1145/2422518.2422525

Landau ID, Lozano R, M'Saad M, Karimi A (2011) Adaptive Control: Algorithms, Analysis and Applications, Communications and Control Engineering. Springer, New York, NY, USA

Malek S, Edwards G, Brun Y, Tajalli H, Garcia J, Krka I, Medvidovic N, Mikic-Rakic M, Sukhatme GS (2010) An architecture-driven software mobility framework. Journal of Systems and Software 83(6):972–989. doi:10.1016/j.jss.2009.11.003

Müller H, Pezzè M, Shaw M (2008) Visibility of control in adaptive systems. In: Proceedings of the 2nd International Workshop on Ultra-large-scale Software-intensive Systems. ULSSIS 2008. ACM, New York, NY, USA. pp 23–26. doi:10.1145/1370700.1370707

Ogel F, Folliot B, Piumarta I (2003) On reflexive and dynamically adaptable environments for distributed computing. In: 23rd International Conference on Distributed Computing Systems Workshops (ICDCS 2003 Workshops), 19-22 May 2003, Providence, RI, USA. IEEE Computer Society, Washington, DC, USA. pp 112–117. doi:10.1109/ICDCSW.2003.1203541

Parunak HVD, Brueckner SA (2011) Software engineering for self-organizing systems. In: Weyns D, Müller JP (eds). 12th International Workshop on Agent-Oriented Software Engineering (AOSE 2011), AAMAS 2011, Taipei, Taiwan

Patikirikorala T, Colman AW, Han J, Wang L (2012) A systematic survey on the design of self-adaptive software systems using control engineering approaches. In: 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012, Zurich, Switzerland, June 4-5, 2012. IEEE, Washington, DC, USA. pp 33–42. doi:10.1109/SEAMS.2012.6224389

Räihä O (2010) A survey on search-based software design. Computer Science Review 4(4):203–249. doi:10.1016/j.cosrev.2010.06.001

Salehie M, Tahvildari L (2009) Self-adaptive software: landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 4(2):14–11442. doi:10.1145/1516533.1516538

Slotine J-JE, Li W (1991) Applied Nonlinear Control. Prentice Hall, Englewood Cliffs (N.J.) http://opac.inria.fr/record=b1132812

Stevens SS (1946) On the Theory of Scales of Measurement. Science 103(2684):677–680. doi:10.2307/1671815

Tilbury DM, Parekh S, Diao Y, Hellerstein JL (2004) Feedback Control of Computing Systems, Wiley interscience publication. Wiley IEEE press, Hoboken, NJ US. http://opac.inria.fr/record=b1119042

van Veldhuizen DA, Lamont GB (1999) Multiobjective evolutionary algorithm test suites. In: SAC. pp 351–357. doi:10.1145/298151.298382

Van Veldhuizen DA, Lamont GB (2000) On measuring multiobjective evolutionary algorithm performance. In: Evolutionary Computation, 2000. Proceedings of the 2000 Congress On. IEEE, Washington, DC, USA Vol. 1. pp 204–211

Vogel T, Giese H (2012) A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In: 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012, Zurich, Switzerland, June 4-5, 2012. IEEE, Washington, DC, USA. pp 129–138. doi:10.1109/SEAMS.2012.6224399. http://dx.doi.org/10.1109/SEAMS.2012.6224399

Vokác M, Tichy WF, Sjøberg DIK, Arisholm E, Aldrin M (2004) A controlled experiment comparing the maintainability of programs designed with and without design patterns – a replication in a real programming environment. Empirical Software Engineering 9(3):149–195

Wang Q (2005) Handbook of PI and PID controller tuning rules, aidan o'dwyer, imperial college press, London, 375pp, ISBN 1-86094-342-x, 2003. Automatica 41(2):355–356. doi:10.1016/j.automatica.2004.09.012

Weyns D, Iftikhar MU, Söderlund J (2013) Do external feedback loops improve the design of self-adaptive systems? a controlled experiment. In: Litoiu M, Mylopoulos J (eds). Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS 2013, San Francisco, CA, USA, May 20-21, 2013. IEEE/ACM, Washington, DC, USA. pp 3–12

Weyns D, Malek S, Andersson J (2012) FORMS: Unifying reference model for formal specification of distributed self-adaptive systems. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 7(1):8

Weyns D, Schmerl BR, Grassi V, Malek S, Mirandola R, Prehofer C, Wuttke J, Andersson J, Giese H, Göschka KM, de Lemos R (2010) On patterns for decentralized control in self-adaptive systems. In: Giese H, Müller HA, Shaw M (eds). Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers. Lecture Notes in Computer Science. Springer, New York, NY, USA Vol. 7475. pp 76–107. doi:10.1007/978-3-642-35813-5_4

White T (2009) Hadoop: the Definitive Guide. O'Reilly Media, Inc., Sebastopol, CA, USA

Wijayasiriwardhane T, Lai R (2010) Component Point: A system-level size measure for component-based software systems. J Syst Softw 83(12):2456–2470. doi:10.1016/j.jss.2010.07.008

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B (2012) Experimentation in Software Engineering. Springer, New York, NY, USA. doi:10.1007/978-3-642-29044-2