A Search-Based Approach for Architectural Design of Feedback Control Concerns in Self-Adaptive Systems

CONFERENCE PAPER · SEPTEMBER 2013

DOI: 10.1109/SASO.2013.42

2 AUTHORS:

Sandro Andrade
Federal Institute of Education, Science, and …
37 PUBLICATIONS **29** CITATIONS
SEE PROFILE

Raimundo J de A Macêdo
Universidade Federal da Bahia
70 PUBLICATIONS **536** CITATIONS
SEE PROFILE

A Search-Based Approach for Architectural Design of Feedback Control Concerns in Self-Adaptive Systems

Sandro S. Andrade[†] [§]
[§]GSORT Distributed Systems Group
Federal Institute of Education, Science
and Technology of Bahia. Salvador-Ba. Brazil
Email: sandros@ufba.br

Raimundo José de A. Macêdo[†]
[†]Distributed Systems Laboratory (LaSiD)
Department of Computer Science
Federal University of Bahia. Salvador-Ba. Brazil
Email: macedo@ufba.br

***Abstract*—A number of approaches for endowing systems with self-adaptive behavior have been proposed over the past years. Among such efforts, architecture-centric solutions with explicit representation of feedback loops have currently been advocated as a promising research landscape. Although noteworthy results have been achieved on some fronts, the lack of systematic representations of architectural knowledge and effective support for well-informed trade-off decisions still poses significant challenges when designing modern self-adaptive systems. In this paper, we present a systematic and flexible representation of design dimensions related to feedback control concerns, a set of metrics which estimate quality attributes of resulting automated designs, and a search-based approach to find out a set of Pareto-optimal candidate architectures. The proposed approach has been fully implemented in a supporting tool and a case study with a self-adaptive cloud computing environment has been undertaken. The results indicate that our approach effectively captures the most prominent degrees of freedom when designing self-adaptive systems, helps to elicit effective subtle designs, and provides useful support for early analysis of trade-off decisions.**

Keywords*—*self-adaptive systems, feedback control, automated software architecture design, design knowledge management

I. Introduction

The design of modern large-scale distributed systems for operating in highly unpredictable and changing environments has brought significant challenges to self-adaptive systems architects [1][2]. Architecture-centric approaches with explicit (first-class) representation of feedback loops [3][4][5] have been advocated as a promising research direction in such a landscape, due to their generality and support for early reasoning of self-adaptation quality attributes.

Although a lot of current research have been devoted to the use of feedback control theory as the enabling mechanism for self-adaptation [6][7], it seems hard – even for experienced architects – to understand and evaluate trade-offs among currently available approaches. Deciding about a specific system modeling approach, control law, tuning method, and arrangements of feedback loops, heavily impacts not only system's self-adaptation robustness and control overhead, but also common quality attributes such as scalability, flexibility and maintainability. Furthermore, the variety of solutions currently available in many design dimensions rapidly open up a huge space of candidate architectures for self-adaptive behavior. Manually designing and evaluating such architectures is an expensive, time-consuming and error-prone task.

In addition, software engineering researchers have been urged to drive their efforts towards an engineering discipline for software [8], which mostly involves the prospection of theories [9][10][11] and organization of knowledge for routine use [12]. In particular, the use of design theories [10] for self-adaptive systems is still on its infancy. The systematic gathering of such engineering knowledge – in terms of handbooks, architecture catalogs, evaluation methods, and supporting tools – would help minimize the burden of acquiring such refined experience and further leverage self-adaptive systems research.

In this paper, we present DuSE[1] – a systematic and flexible representation of architectural design dimensions, its accompanying architecture design process, and the DuSE-MT supporting tool. For that purpose, we have defined a model-based approach for design spaces representation and exploration which entails: i) a five-layer metamodeling stack enabling automated architecture redesign; ii) a general framework for architectural metrics support; and iii) a search-based mechanism which explicitly points out decision trade-offs (a set of Pareto-optimal candidate architectures).

From such underlying mechanism for design spaces representation, we have designed and implemented SA:DuSE – a specific DuSE instance which defines design dimensions and evaluation metrics devoted to feedback control in self-adaptive systems. SA:DuSE captures prominent degrees of freedom when designing self-adaptive systems such as the adopted control law, tuning approach, variations in control loop deployment, and control adaptability. We systematically represent the architectural changes that must be enacted, in an initial (non-adaptive) architecture model, in order to yield a particular self-adaptation solution. The metrics defined in SA:DuSE evaluate resulting architectures regarding aspects related to control overhead, control robustness, settling time and overshoot.

As a consequence of a wide range of available approaches for feedback control of software systems, such systematic representation of design spaces and architecture redesign changes rapidly opens up a huge set of resulting candidate architectures.

[1] http://duse.sf.net

For instance, when analyzing a small three controllable components application, SA:DuSE generates 54,010,152 resulting candidate architectures, each one favoring a specific evaluation metric. With a four controllable components application, the number of resulting architectures rapidly increases to 20,415,837,000. That motivates the use of a multi-objective optimization approach [13][14], which allows architects to focus on those solutions differing only in the metric they favor (Pareto-front) and, therefore, explicitly eliciting a trade-off decision. DuSE-MT implements the Pareto-front discovery for any DuSE instance, minimizing the burden of manual design space exploration and supporting well-informed decisions.

We have evaluated SA:DuSE capabilities in a case study aimed at providing self-adaptive behavior to a cloud-based media encoding service. In such scenario, stringent user requirements for Quality of Service (QoS) and Service Level Agreements (SLAs), in addition to provider requirements for cost savings, typically require the use of mechanisms for automatic feedback control. The results indicate our approach effectively helps to elicit effective subtle solutions and provides useful support for early analysis of trade-off decisions.

The remainder of this paper is organized as follows. Section II presents current challenges when conveying software architecture knowledge. The DuSE domain-independent metamodel is described in section III, while section IV presents the DuSE instance devoted to feedback control concerns. Details about the multi-objective optimization approach we used to elicit design trade-offs are discussed in section V. Section VI describes the case study we have undertaken to evaluate SA:DuSE capabilities and section VII provides pointers to related work. Finally, we discuss our solution and present some avenues for future research in section VIII.

II. Systematic Conveying of Self-Adaptive Systems Architecture Design Knowledge

A software architecture manifests major early design decisions which not only guide the system's development and deployment but also directly determine the resulting quality attributes. It has been argued [12] that the availability of systematically structured architectural knowledge can greatly improve the software design process. The nuances and idiosyncrasies of self-adaptive systems development readily make the required design skills, if not managed, become tacit knowledge which is eroded as architects leave.

Over the past years, a number of technocratic and behavioral approaches for managing architectural knowledge have been proposed [12]. Most promising efforts include the definition of architecture styles catalogs, reference architectures, domain-specific design processes, and architectural knowledge repositories. In this paper, we focus on defining a systematic and flexible representation of design spaces [15][16] to leverage the capture of implicit architecture knowledge. Whilst no single approach would be able to cope with all involved facets of architectural knowledge, we believe that comprehensive design spaces representations along with search-based approaches for automated architecture design constitute a promising research front.

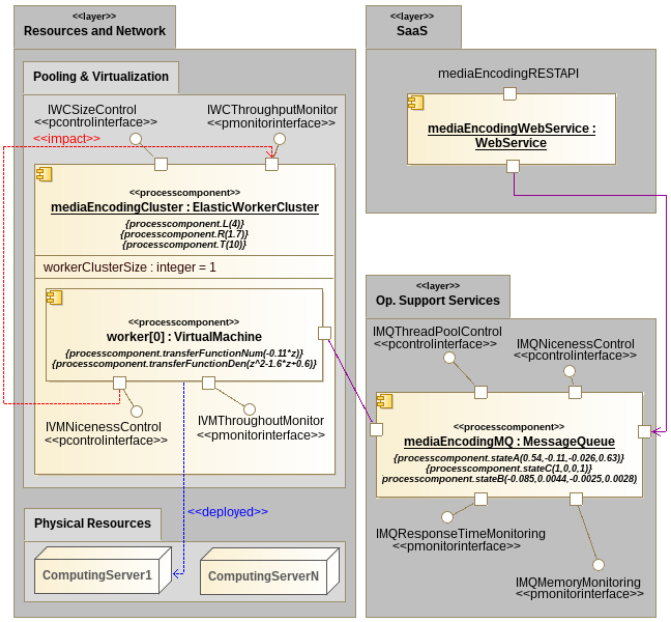

Fig. 1. A cloud-based media encoding service. Encoding jobs are submitted from the `mediaEncodingWebService` component and stored in the `mediaEncodingMQ` message queue for asynchronous processing. A cluster of working virtual machines is managed by the cloud infrastructure. Although interfaces for managing resource allocation are already available, there are no automatic feedback control loops deployed in the cloud.

A. Reasons for a Search-Based Approach

Designing complex software-intensive systems demands the systematic use of refined experience, appropriate tools, and sophisticated decision making skills. False intuition and technology bias are often the causes of major shortcomings in software architectures, leading to unintended quality properties, implicit assumptions, and easily eroding and aging solutions. Search-Based Software Engineering (SBSE) [17] techniques have successfully been addressing this problem by automatically scouring the search space for the solutions that best fit a given objective function.

In particular, a lot of search-based approaches for software architecture design [18] have been proposed over the last years. The demands for heavily specialized design skills, the wide range of techniques for feedback control currently available, and the lack of systematically structured architecture design knowledge when developing self-adaptive systems have been the major motivating factors of this research.

B. Motivating Example

Self-adaptive systems have played a crucial role in many domain applications, including data center automation, real-time systems, cloud computing, middleware, databases, and embedded systems [6].

Fig. 1 depicts the structural (Component & Connector) architecture view for a cloud-based media encoding service. Increasing processing costs of encoding media for a number of combinations of target devices and network bandwidths have currently motivated the use of scalable cloud-based architectures in such an application domain. Cloud Service Providers should be able to enforce SLAs so that Media Providers readily

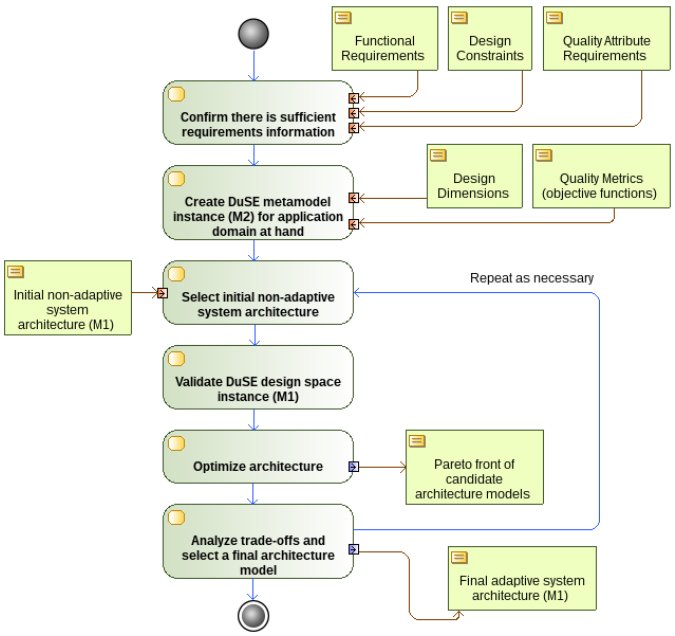

Fig. 2. The DuSE Architecture Design Process. Our search-based design approach scours the search space for solutions that best fit defined quality attributes. DuSE run-time infrastructure supports any specific DuSE design space instance.

make all required encodings for latest film releases available. In order to save costs and optimize resource allocation, it is mandatory to adopt some form of feedback control loop to automatically manage the cloud infrastructure.

Each worker virtual machine (VM) encodes a specific subset of a movie's frames and may have its niceness value (operating system task's priority) modified (`IVMNicenessControl` interface) in order to control the perceived partial encoding throughput (`IVMThroughputMonitor` interface). Should individual VMs' niceness control not be enough to enforce the involved SLAs, new VMs instances may be started by manipulating the `IWCSizeControl` interface on `mediaEncodingCluster` component. The cluster-wide global encoding throughput is provided by the `IWCThroughputMonitor` interface.

Serving multiple worker VMs concurrently may easily cause the `mediaEncodingMQ` message queue to become a central bottleneck. To enable the fine-tuning of message queue's performance, its average response time and memory consumption can be controlled by manipulating its `IMQThreadPoolControl` and `IMQNicenessControl` provided interfaces.

Such simple but realistic scenario, with its demands for mandatory self-adaptation capabilities, is already complex enough to challenge architects with a large, highly specialized, and quality-varying solution space. The choices for a particular control law, different schemas for global/hierarchical/decentralized feedback loops, and approaches for robust control are only some of the dimensions that directly influences the resulting architecture's quality attributes.

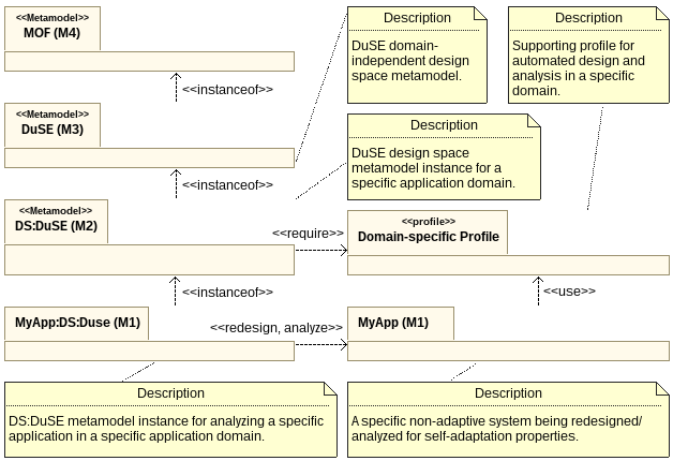

Fig. 3. DuSE Model-Based Architecture Overview. A five-layer metamodeling stack defines the underlying mechanism that supports DuSE architecture design process.

C. The DuSE Architecture Design Process

The aforementioned self-adaptive systems design challenges have motivated the proposal of DuSE – our search-based approach for domain-independent representation of design spaces and automated well-informed architecture design. A specific instance of DuSE – SA:DuSE – systematically captures major design dimensions and architecture evaluation metrics regarding feedback control concerns.

The proposed architecture design process, shown in Fig. 2, requires that a particular domain-specific DuSE instance – such as SA:DuSE – be previously created. Furthermore, an architecture structural view for an initial system should be available. Such architectural annotations rely on an UML Profile which accompanies the domain-specific DuSE instance and provide minimal information about initial system characteristics.

The SA:DuSE accompanying UML Profile we have defined introduces major stereotypes and tagged values related to self-adaptation aspects. Some of those artifacts are shown in Fig. 1. Due to space limitations, we timely discuss such annotations briefly along the section IV. The initial system architecture represents the non-adaptive system to be endowed with self-adaptation capabilities.

A five-layer metamodeling stack, depicted in Fig. 3, defines the underlying mechanism that supports DuSE architecture design process. DuSE acts as a $M3$ layer metamodel (language) for description of architectural design spaces. It is described in MOF (Meta Object Facility) which, in our architecture, acts as a $M4$ layer metamodel. Specific DuSE instances – domain-specific design spaces and metrics like SA:DuSE – act as $M2$ layer metamodels. Initial system architectures, annotated with specific UML Profiles, act as $M1$ layer models and will be automatically redesigned by DuSE run-time infrastructure, yielding a set of $M1$ layer candidate architectures. When using SA:DuSE, such candidate architectures provide alternative solutions for self-adaptation. The actual (running) self-adaptive system instances (not shown in figure) lie in $M0$ layer.

Initial $M1$ layer system architecture models, provided as input to DuSE design process, usually contain multiple *loci*

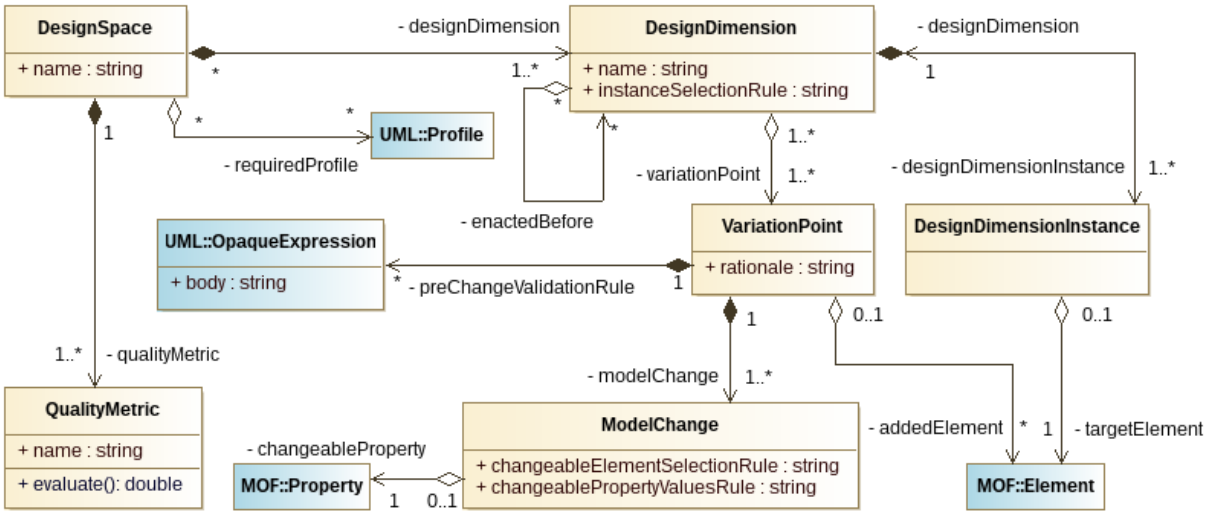

Fig. 4. DuSE Domain-Independent Design Space Metamodel (simplified). A design space is represented as a tuple DS= $<DD, M>$, where *DD* is a set of design dimensions and *M* is a set of metrics defined for that design space. A design dimension DD_i contains a set of variation points VP_{ij}=$<CT_{ij}, CH_{ij}>$, where CT_{ij} is a set of OCL constraints that must be satisfied in order to that design dimension point be available for use, and CH_{ij} is the set of architecture changes that a given dimension contributes to the final candidate architecture. A particular design dimension may yield multiple design dimension instances because of multiple *loci* of similar decisions in the initial input model. A partial order for design dimension evaluation supports dependent dimensions.

where the same sort of architecture decision should be taken. For instance, in SA:DuSE input models, the existence of two controllable components require that all decisions (design dimensions) regarding feedback control aspects be individually taken for each component. In order to support that, a domain-specific DuSE instance – e.g. SA:DuSE – must be instantiated again, so that design dimensions are properly replicated for each identified decision *locus*. That input-specific SA:DuSE instance is shown in Fig. 3 as the $M1$ layer *MyApp:DS:DuSE* model. Such input-specific design space is actually where our search-based approach looks for candidate architectures.

III. Domain-Independent Design Space Metamodel

In [19] we presented DuSE and a preliminary version of a manually explorable SA:DuSE design space. This paper builds on that work by defining a more expressive – but also harder to explore – design space for feedback control concerns. Furthermore, we extend that previous work with the proposed multi-objective optimization architecture design approach.

Fig. 4 presents the DuSE domain-independent metamodel for representing design spaces and evaluation metrics. A design space is defined as a tuple DS= $<DD, M>$, where *DD* is a set of design dimensions and *M* is a set of metrics defined for that design space. A design dimension DD_i is a set of variation points VP_{ij}=$<CT_{ij}, CH_{ij}>$, where CT_{ij} is a set of OCL constraints and CH_{ij} is a set of changes to be enacted in the target system's original model SM_0. The OCL constraints define what must be satisfied (evaluated as true) in the target system's architecture model, so that the corresponding variation point be available for use.

Design dimensions are seldom independent. Such dependencies come out in two different constructions: invalid variation point combinations and previous evaluation dependencies. Some design space locations may define invalid solutions because of conflicting architectural changes provided by the involved variation points. A design dimension may depend upon a previous evaluation of related pre-requisite design dimensions. For instance, enacting architectural changes regarding control loop tuning requires the previous enacting of changes related to control law variation points. DuSE metamodel allows for defining such dependencies. The proposed search-based design approach generates a topological order for design dimensions evaluation and DuSE's OCL well-formedness rules certify there are no cycles in dimension dependencies.

A specific design space location $<VP_{1i}, VP_{2j}, ..., VP_{|DD|m}>$ is a valid solution for an initial model SM_0 only if the partially modified resulting model satisfy the set of constraints $CT_{1i} \cup CT_{2j} \cup ... \cup CT_{|DD|m}$, evaluated in the topological order derived from design dimension dependencies. Likewise, the changes to be enacted in the original model SM_0 for a given design space location $<VP_{1i}, VP_{2j}, ..., VP_{|DD|m}>$ are the merge of all architectural contributions from each design dimension location: $CH_{1i} \cup CH_{2j} \cup ... \cup CH_{|DD|m}$. We denote by $SM_{\{i,j,...,m\}}$ the architectural model resulting after applying contributions from all specific locations in each design space dimension.

DuSE initial input models may require multiple evaluation of a single design dimension, applied to different architectural *loci* but that demands the same sort of design decision captured by such dimension. In order to support the automatic discovery of such decision *loci* – and the creation of corresponding design dimension instances – design dimension's `instanceSelectionRule` attribute holds an OCL expression which returns a collection of model elements demanding that design dimension resolution. Such expressions usually look for annotated arrangements of components/connectors, supported by the design space accompanying UML Profile. An architectural model change is defined in terms of the input metamodel's property to be modified for those already existing elements and/or model elements to be added to the original model.

An architecture evaluation metric M_i is a tuple

$<ME_i, MG_i>$, where ME_i is the metric evaluation expression and MG_i describes if the metric is intended to be maximized (1) or minimized (-1). DuSE run-time infrastructure provides the means to evaluate all metrics for any valid design space location, setting the stage for an automated search-based design approach.

IV. Design Space Instance for Self-Adaptive Systems Domain

In order to support early reasoning and analysis of feedback control approaches for self-adaptation, we have instantiated DuSE in terms of design dimensions, architectural changes, metrics, and constraints directly related to most prominent feedback control concerns. Table I describes SA:DuSE, our domain-specific design space for feedback control concerns in self-adaptive systems.

It is worth mentioning that, while we expect such dimensions expressively represent a wide range of design alternatives, SA:DuSE is not intended to be an exhaustive enumeration of degrees of freedom when designing self-adaptive systems. SA:DuSE has been continuously evolved, in an attempt to elicit representative design dimensions and effective evaluation metrics for self-adaptive systems. Currently, SA:DuSE entails four design dimensions and four evaluation metrics. SA:DuSE design dimensions encompass aspects regarding control laws, loop tuning approaches, control adaptation techniques, and loop deployment. SA:DuSE provides evaluation metrics related to control operation overhead, average settling time, average maximum overshoot, and control robustness.

A. Design Dimensions

SA:DuSE currently entails the following dimensions:

1) DD_1 – Control Law: captures the most prevalent algorithms and mechanisms which decide how much control to enact in the controllable component.

a) Variation Points: dimension DD_1 defines some of the most widely used approaches for SISO (*Single-Input Single-Output*) and MIMO (*Multiple-Input Multiple-Output*) control. That includes variations of fixed-gain feedback control (Proportional, Proportional-Integral, and Proportional-Integral-Derivative) and state space feedback control (Static, Precompensated Static, and Dynamic), respectively.

b) Design Dimension Instance Selection Rules: DuSE run-time infrastructure will create a DD_1 instance for each component annotated with stereotype `<<ParametricProcessComponent>>` or its derivatives: components described by ARX (*Auto-Regressive with eXogenous input*), FOPDT (*First-Order Plus Dead Time*), and SS (*State Space*) models.

c) Pre-Change OCL Validation Rules: validation rules for variation points in this dimension aim at certifying that the dimension instance's target element is likely to be controlled by using the represented control law. For instance, controllable components described by SS models must not be susceptible to fixed-gain control laws and, therefore, make invalid such variation points and any involved candidate architecture.

2) DD_2 – Tuning Approach: captures a range of empirical tuning methods [20] available for controlling components whose models are derived from parameters gathered in a *bump test*, a step change in the control input. This dimension defines a dependency to DD_1 and therefore should be evaluated after all DD_1 architectural changes have already been enacted in the initial model.

a) Variation Points: dimension DD_2 defines the following tuning methods: *i)* CHR (*Chien-Hrones-Reswick*) tuning with variants for 0% (CHR-0OS-*) or 20% (CHR-20OS-*) maximum overshoot and emphasizing disturbance rejection (CHR-*-DR) or reference tracking (CHR-*-RT); *ii)* Ziegler-Nichols (ZN); *iii)* Cohen-Coon (CC); and *iv)* LQR (*Linear Quadratic Regulator*) optimal control tuning.

b) Design Dimension Instance Selection Rules: same from DD_1, since every controlled component has a control loop that must be tuned (off-line or on-line).

c) Pre-Change OCL Validation Rules: rules in this dimension aim at certifying that the control law selected in DD_1 is likely to be tuned by using the represented tuning method. For instance, LQR tuning method requires the use of SS models and, therefore, makes invalid solutions involving ARX and FOPDT models.

3) DD_3 – Control Adaptation: highly restrictive assumptions when modeling/tuning control loops and nonlinearities present in computing systems have motivated the adoption of adaptive control approaches [21]. This dimension captures two widely used solutions for enhancing control robustness: gain scheduling and MIAC (*Model Identification Adaptive Control*). This dimension defines a dependency to DD_1.

a) Variation Points: dimension DD_3 defines three levels of control adaptation: fixed-gain (actually representing a non-adaptive control), gain scheduling (for predictable variations in process dynamics), and MIAC (for unpredictable variations in process dynamics – on-line tuning).

b) Design Dimension Instance Selection Rules: same from DD_1, since every controlled component has a control loop that holds some level of robustness.

c) Pre-Change OCL Validation Rules: gain scheduling variation point requires the design dimension instance's target element holds one system model for each predictable operating region. That is indicated by annotating the component with a `<<GSProcessComponent>>` stereotype.

4) DD_4 – MAPE Deployment: Control loops have currently been described in terms of its constituent MAPE components: Monitor, Analyze, Plan and Execute. Interacting control loops and different deployment models of its constituent MAPE components also directly impacts the resulting architecture quality attributes. This dimension captures three commonly used MAPE deployment patterns ([7], sec. 4.6.3, pag. 125): global control, local control + shared reference, and local control + shared error. This dimension defines a dependency to DD_3.

a) Variation Points: dimension DD_4 defines three patterns for MAPE deployment: *i)* global control; *ii)* local control + shared reference; and *iii)* local control + shared error. The adoption of different deployment patterns usually incurs a trade-off between control performance and control overhead.

b) Design Dimension Instance Selection Rules: same from DD_1, since every controlled component has a control loop that must be deployed in accordance with some deployment pattern.

c) Pre-Change OCL Validation Rules: similar to those defined in DD_1, but here checking for a generic controller.

B. Quality Metrics

Table II presents the domain-specific evaluation metrics currently available in SA:DuSE. Although domain-independent metrics related to static structural aspects, such as cohesion and coupling, often provide useful insight when evaluating software architectures, here we focus on metrics devoted to the specific domain of self-adaptive systems. Such metrics may be evaluated by DuSE run-time infrastructure for any valid resulting candidate architecture. The metrics rely on specific annotations (use of the domain-specific UML Profile) found in the resulting candidate architecture under evaluation.

Metric M_1 denotes how much system-wide control overhead are going to incur in the candidate architecture. SA:DuSE `QController::overhead()` operation increasingly penalizes architectures yielded from variation points related to adaptive control, as well as those solutions with intensive use of `<<distributor>>` connectors, as a consequence of the use of distributed/decentralized loop deployment patterns.

Metric M_2 builds upon standard control theory analysis techniques to estimates the average settling time incurred from the specific combination of control law and tuning approach found in the candidate architecture.

Metric M_3, also based on control theory analysis techniques, estimates the average maximum overshoot expected in the candidate architecture. As it is well-known from the control theory literature, M_2 and M_3 are usually conflicting goals demanding, therefore, a judicious trade-off analysis.

Finally, metric M_4 denotes to which extent control performance can be guaranteed should the system dynamics deviate from those used when tuning the control loops. Highly unpredictable execution environments and/or stochastic disturbances usually require some sort of control robustness, usually to the detriment of low control overhead.

It is worth mentioning that M_1 and M_4 have no predictive intentions. They denote dimensionless quantities and, therefore, work as relative metrics when comparing candidate architectures.

V. The Search-Based Approach

As we mentioned before, even small input architecture models incur the generation of huge design spaces. For instance, inputs containing three *loci* of control yield 54,010,152 resulting candidate architectures, including invalid ones. That figure rapidly increases to 20,415,837,000 for inputs exhibiting four *loci* of control.

Manually searching such design spaces is a tedious, obtuse, and time-consuming task. Furthermore, it is generally more productive to focus on those candidate architectures differing only in which quality metric they favor. In other words, we are interested in finding out a set of solutions for which it is impossible to make any one architecture better off without make at least another one worse. That is an informal definition of Pareto-front [13] – a common solution provided by *a posteriori* multi-objective optimization solvers.

```
1:  procedure DUSE_OPT(i, di)
2:                                  ▷ i = input model; di = DuSE instance
3:      s ← globalSettings()
4:      dii ← createAppDSInstance(i, di)
5:      p ← randomPopulation(dii, s.populationSize)
6:      curIteration ← 0
7:      while curIteration ≤ s.maxIterations do
8:          p ← nsga2Rank(p, dii, s.populationSize)
9:          p ← p ∪ nsga2Mate(p, s.offspringSize)
10:         curIteration ← curIteration + 1
11:     end while
12:     return nsga2highestParetoRank(p)
13: end procedure
```

Fig. 5. The DuSE multi-objective optimization algorithm (simplified). DuSE implements a NSGA-II-based elitist evolutionary approach for multi-objective optimization.

Finding the exact globally Pareto-front in huge search spaces usually incur in prohibitive costs. Fortunately, near-optimal Pareto-fronts – found in reasonable time, for instance, by metaheuristic-based approaches – are often enough to elicit interesting architectural trade-offs. DuSE builds upon an elitist evolutionary algorithm [13] to find out a near-optimal Pareto-front of candidate architectures.

A. The DuSE Multi-Objective Optimization Algorithm

Elitist evolutionary algorithms for multi-objective optimization are a class of population-based metaheuristics supposed to converge faster than its non-elitist counterparts. The DuSE multi-objective optimization algorithm, depicted in Fig. 5, builds upon the elitist evolutionary optimization approach NSGA-II (*Nondominated Sorting Genetic Algorithm II*) [22].

Initially (step 4), the application-specific SA:DuSE design dimension instance is generated by DuSE, creating the required design dimension instances for all *loci* of architectural decisions found in the input model. Then, DuSE defines the initial population by randomly selecting a set of `populationSize` valid candidate architectures (step 5). Next, the NSGA-II-based evolutionary iterations (steps 7-11) repeatedly wage new architecture candidates (the offspring) and select the more promising solutions (according to the NSGA-II defined Pareto rank) to take part in the next generation. The process continues until the maximum number of iterations is hit, returning the highest ranked Pareto-front from the latest population.

DuSE run-time infrastructure automatically converts candidate UML models to solution vectors containing the design space location that yields such candidate. Such vectors enabled the implementation of the DuSE *selection*, *crossover* and *mutation* operators which support the use of an evolutionary approach.

VI. Evaluation

The approach we propose in this paper has been fully implemented in DuSE-MT: the DuSE architecture design process suporting tool. DuSE-MT enables the process depicted

TABLE I. SA:DUSE – DUSE DESIGN SPACE METAMODEL INSTANCE FOR FEEDBACK CONTROL CONCERNS IN SELF-ADAPTIVE SYSTEMS

Design Dimension	Variation Points	Pre-Change OCL Validation Rules	Model Changes
(DD_1) Control Law	(VP_{11}) Proportional	(CT_{111}) *target.extension_SISOProcessComponent ->notEmpty()*	(CH_{111}) *target.namespace.addOwnedType(s = sensorFactory("QSISOSensor", target.mInterface()));* (CH_{112}) *target.namespace.addOwnedType(a = actuatorFactory("QSISOActuator", target.cInterface()));* (CH_{113}) *target.namespace.addOwnedType(c = controllerFactory("QPController", s, a));*
	(VP_{12}) Proportional-Integral	(CT_{121}) *target.extension_SISOProcessComponent ->notEmpty()*	CH_{121} and CH_{122} as defined in CH_{111} and CH_{112} (CH_{123}) *target.namespace.addOwnedType(c = controllerFactory("QPIController", s, a));*
	(VP_{13}) Proportional-Integral-Derivative	(CT_{131}) *target.extension_SISOProcessComponent ->notEmpty()*	CH_{131} and CH_{132} as defined in CH_{111} and CH_{112} (CH_{133}) *target.namespace.addOwnedType(c = controllerFactory("QPIDController", s, a));*
	(VP_{14}) Static State Feedback	(CT_{141}) *target.extension_MIMOProcessComponent ->notEmpty()*	(CH_{141}) *target.namespace.addOwnedType(s = sensorFactory("QMIMOSensor", target.mInterfaces()));* (CH_{142}) *target.namespace.addOwnedType(a = actuatorFactory("QMIMOActuator", target.cInterfaces()));* (CH_{143}) *target.namespace.addOwnedType(c = controllerFactory("QSSController", s, a));*
	(VP_{15}) Precompensated Static State Feedback	(CT_{151}) *target.extension_MIMOProcessComponent ->notEmpty()*	CH_{151} and CH_{152} as defined in CH_{141} and CH_{142} (CH_{153}) *target.namespace.addOwnedType(c = controllerFactory("QPSSController", s, a));*
	(VP_{16}) Dynamic State Feedback	(CT_{161}) *target.extension_MIMOProcessComponent ->notEmpty()*	CH_{161} and CH_{162} as defined in CH_{141} and CH_{142} (CH_{163}) *target.namespace.addOwnedType(c = controllerFactory("QDSController", s, a));*
(DD_2) Tuning Approach	(VP_{21}) CHR-0OS-DR	(CT_{211}) *target.extension_ProcessComponent .oclIsTypeOf (FOPDTProcessComponent)* (CT_{212}) *target.type.oclAsType(EncapsulatedClassifier).ownedPort-> select(provided->exists(extension_PMonitorInterface-> notEmpty())) .end.definingEnd.type-> exists(extension_Controller.oclIsKindOf (SISOController))*	(CH_{211}) *if (target.appliedController() instanceof QPController) { appliedController.setParameters(0.3/target.extension_ProcessComponent.a, 0.0, 0.0); } else if (target.appliedController() instanceof QPIController) { target.appliedController().setParameters(0.6/target.extension_FOPDTProcessComponent.a, 4*target.extension_FOPDTProcessComponent.L, 0.0); } else if (target.appliedController() instanceof QPIDController) { target.appliedController().setParameters(0.95/target.extension_FOPDTProcessComponent.a, 2.4*target.extension_FOPDTProcessComponent.L, 0.42*target.extension_FOPDTProcessComponent.L); }*
	(VP_{22}) CHR-0OS-RT	CT_{221} and CT_{222} as defined in CT_{211} and CT_{212}	CH_{221} as defined in CH_{211} but using CHR-0OS-RT tuning
	(VP_{23}) CHR-20OS-DR	CT_{231} and CT_{232} as defined in CT_{211} and CT_{212}	CH_{231} as defined in CH_{211} but using CHR-20OS-DR tuning
	(VP_{24}) CHR-20OS-RT	CT_{241} and CT_{242} as defined in CT_{211} and CT_{212}	CH_{241} as defined in CH_{211} but using CHR-20OS-RT tuning
	(VP_{25}) Ziegler-Nichols	CT_{251} and CT_{252} as defined in CT_{211} and CT_{212}	CH_{251} as defined in CH_{211} but using Ziegler-Nichols tuning
	(VP_{26}) Cohen-Coon	CT_{261} and CT_{262} as defined in CT_{211} and CT_{212}	CH_{261} as defined in CH_{211} but using Cohen-Coon tuning
	(VP_{27}) LQR	CT_{271} and CT_{272} as defined in CT_{211} and CT_{212} but checking for a State Space process component and a MIMO controller	CH_{271} as defined in CH_{211} but using LQR optimal tuning
(DD_3) Control Adaptation	(VP_{31}) Fixed-Gain (no adaptation)	CT_{311} as defined in CT_{111}	<no changes required>
	(VP_{32}) Gain Scheduling	(CT_{321}) *target.extension_ GSProcessComponent->notEmpty()*	(CH_{321}) target.namespace.addOwnedType(s = new QGainScheduler(target, c));
	(VP_{33}) MIAC	CT_{331} as defined in CT_{111}	(CH_{331}) target.namespace.addOwnedType(e = new QModelEstimator(target, c));
(DD_4) MAPE Deployment	(VP_{41}) Global Control	CT_{411} and CT_{412} as defined in CT_{211} and CT_{212} but checking for a generic process component and a generic controller	(CH_{411}) device d = new QUmlDevice; (CH_{412}) c.setDeploymentNode(d); (CH_{413}) s.setDeploymentNode(d); (CH_{414}) a.setDeploymentNode(d);
	(VP_{42}) Local Control + Shared Reference	CT_{421} and CT_{422} as defined in CT_{411} and CT_{412}	(CH_{421}) c.setDeploymentNode(target.deploymentNode()); (CH_{422}) s.setDeploymentNode(target.deploymentNode()); (CH_{423}) a.setDeploymentNode(target.deploymentNode());
	(VP_{43}) Local Control + Shared Error	CT_{431} and CT_{432} as defined in CT_{411} and CT_{412}	(CH_{431}) c.setMAPEConfig(QController::E); (CH_{432}) target.namespace.addOwnedType(cc = controllerFactory(c)); (CH_{433}) cc.setMAPEConfig(QController::MAP); (CH_{434}) device d = new QUmlDevice; (CH_{435}) cc.setDeploymentNode(d); (CH_{436}) c.setDeploymentNode(target.deploymentNode());

TABLE II. SA:DuSE Domain-Specific Metrics for Architectural Quality Attributes

Description	Evaluation Expression	Goodness Factor
(M_1) Control Overhead	$ME_1 = \frac{allOwnedElements()->selectAsType(QController)->collect(overhead())->sum()}{allOwnedElements()->selectAsType(QController)->size()}$; $QController::overhead()$ increasingly penalizes (VP_{32}), (VP_{33}), (VP_{41}) and (VP_{43})	$MG_1 = -1$
(M_2) Average Settling Time	$ME_2 = \frac{allOwnedElements()->selectAsType(QParametricController)->sum(stime())}{allOwnedElements()->selectAsType(QParametricController)->size()}$; where $QParametricController::stime() = -4/log(max_i\|p_i\|)$, and $max_i\|p_i\|$ is the magnitude of the largest closed-loop pole	$MG_2 = -1$
(M_3) Average Maximum Overshoot	$ME_3 = \frac{allOwnedElements()->selectAsType(QParametricController)->sum(maxOvershoot())}{allOwnedElements()->selectAsType(QParametricController)->size()}$; where $QParametricController::maxOvershoot() = \begin{cases} 0 & real\ dominant\ pole\ p_1 \geq 0 \\ \|p_1\| & real\ dominant\ pole\ p_1 < 0 \\ r^{\pi/\|\theta\|} & dominant\ poles\ p_1, p_2 = r \cdot e^{\pm j \cdot \theta} \end{cases}$	$MG_3 = -1$
(M_4) Control Robustness	$ME_4 = \frac{allOwnedElements()->selectAsType(QController)->collect(robustness())->sum()}{allOwnedElements()->selectAsType(QController)->size()}$; $QController::robustness()$ increasingly penalizes (VP_{31}) and (VP_{32})	$MG_4 = 1$

in Figure 2 by defining a flexible architecture which allows the use of connector plugins for system identification in a variety of platforms. Each connector plugin enables the probing of target systems developed for that specific platform, gathering input/output relationships between system's controlled/measured parameters and allowing off-line and on-line system identification.

DuSE-MT, shown in Fig. 6, was implemented in C++/Qt and relies on the *Qt Modeling Framework* for handling of UML models described in XMI (*XML Metadata Interchange*); and on *Qt Optimization* for undertaking the evolutionary optimization[2].

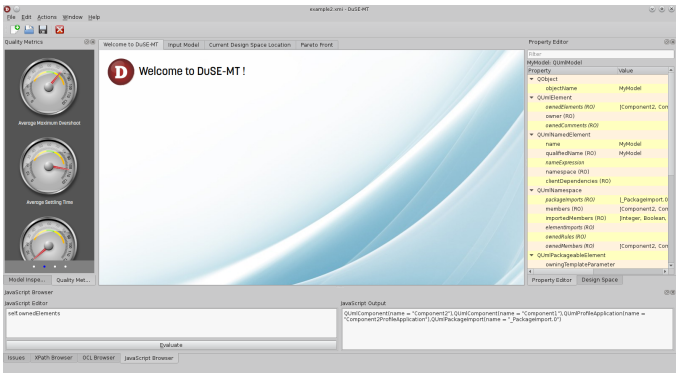

Fig. 6. DuSE-MT: the DuSE supporting tool. Major features include: *i)* definition of new DuSE instances and related metrics; *ii)* optimization of XMI serialized input model with SA:DuSE; and *iii)* visualization of quality metrics.

A. Case Study Set-up

In order to evaluate SA:DuSE regarding its expressiveness when representing self-adaptive systems design decisions and its ability to elicit subtle effective resulting architectures, a case study with the cloud-based motivating example presented in subsection II-B has been undertaken.

For that purpose, an initial UML model for the media encoding service has been created, annotated with system dynamics gathered from experiments, and serialized in XMI format. As mentioned before, such input model generates a design space instance containing 54,010,152 candidate architectures, including invalid ones.

We undertook several optimization runs with different 90 initial random candidates, $populationSize = 60$, $maxIterations = 300$, $offspringSize = 60$, $mutationRate = 0.5$, and $crossoverRate = 0.9$. Binary tournament with crowding distance selection, single-point crossover and bit-wise mutation operators were used in the optimization stage [13]. Each optimization run evaluated around 4900 different candidates and took about 48 seconds on one Intel Core i7, 8Gb RAM, running a x86_64 GNU/Linux 3.9.2 kernel.

[2]http://qt-project.org/wiki/QtModeling - http://gitorious.org/qtoptimization

B. Optimization Outcomes and Threats to Validity

Fig. 7 shows one of the resulting Pareto-fronts generated in the case study. Each pair of quality metrics are compared to each other. Each scatter plot shows the quality property described in its column on the x axis and the quality property in its row on the y axis. The results indicate a major trade-off between metrics M_1 (control overhead) and M_4 (control robustness). We found that M_2 (average settling time) and M_3 (average maximum overshoot) also constitute a design trade-off. Furthermore, the Pareto-front between M_1 and M_4 seems to explicitly elicit the control robustness vs. control overhead trade-off when moving from Fixed-Gain solutions to Gain Scheduling, and then to MIAC.

The Pareto-fronts in such cases smoothly differentiate such quality attributes, indicating that DuSE optimization algorithm indeed converged to a (local) optimum in the case study. The candidate architectures found in Pareto-front M_1/M_4 include the following options for the `mediaEncodingCluster` controller: PI control with CHR-20OS-DR, PID control with CHR-0OS-DR, and P control with Ziegler-Nichols.

Some threats to validity may be identified in the undertaken case study. Conclusion validity threats may be mitigated by the use of techniques to account for random variation and repre-

sentation of formal hypothesis. Further investigation about the impact of NSGA-II parameters on resulting architectures and additional case studies in other self-adaptation scenarios would help to minimize internal validity threats. Specific experiments to evaluate the expressiveness of the proposed metrics and SA:DuSE UML Profile are needed to overcome construct validity threats. The undertaking of controlled experiments using DuSE capabilities, the design of a DuSE *Architecture Implementation Framework* to support automatic artifact generation, and mechanisms for mapping design space exploration to domain-specific design theories are some research fronts that may help mitigate external validity threats.

VII. Related Work

Over the past years some approaches for early architectural reasoning by explicit modeling of control loops have been proposed. In [23], a reference model for self-adaptation (FORMS) is presented. FORMS provides an unified view which integrates perspectives from computational reflection, distributed computing, and MAPE-K [1] technologies. FORMS elements are described in Z specification language, which enables the formal reasoning of self-adaptation properties.

The use of megamodels at run-time to describe self-adaptive behavior is presented in [24]. The proposed notation entails the definition of multiple interacting feedback loops and relies on a model interpreter to dynamically adjust the adaptation logic. An UML Profile for feedback control modeling is presented in [4], along with a set of well-formedness rules to validate control schemas. Guidelines for modeling of multiple loops and elicitation of loop interferences are also presented. Actor-based approaches [25], feature-based mechanisms with on-line learning capabilities [26], and notations for self-adaptation with multiple objectives [27] have also been proposed.

Although some of the aforementioned proposals – such as FORMS – provide expressive and rigorous notations for feedback control modeling, they are mostly based on non-standardized and/or low-parsimony languages, provide no tool support, and still heavily depends on architect's skills, as a consequence of the lack of explicit design space representations.

Our work tries to overcome some of these shortcomings by proposing an architectural analysis environment based on MOF (*Meta Object Facility*) and UML (*Unified Modeling Language*) standards, an explicit domain-specific design space representation, and a set of effective metrics to assess adaptation-related architectural quality attributes. Finally, we observe that our tool leverages rapid modeling/analysis and the proposed design space representation is also liable to be applied in other application domains since we aim at a proper balance between modeling notation generality and expressiveness.

A search-based automated approach for improving performance of software architectures is proposed in [28]. In [29], the authors present a meta-framework for design space exploration. Although our work shares research directions and optimization mechanisms adopted in such efforts, the present paper contributes by investigating the suitability of search-based software architecture design approaches in the particular domain of self-adaptive systems.

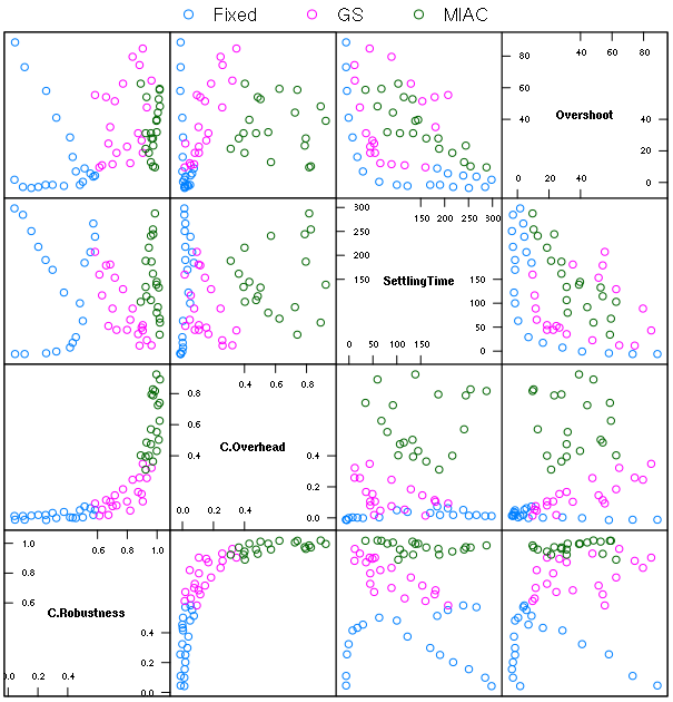

Fig. 7. Scatter plot matrix of a resulting Pareto-front in the cloud-based media encoding case study, showing a pairwise comparison of each quality metric. Each scatter plot shows the quality property described in its column on the x axis and the quality property in its row on the y axis.

VIII. Conclusion and Future Work

The design and development of large-scale distributed systems with flexible and robust self-adaptation capabilities have become a promising approach to cope with continuous increases in system complexity. Furthermore, stringent demands to provide quality of service assurances in unpredictable and uncertain environments introduce additional challenges in such scenarios.

In this paper, we have presented DuSE – a flexible design and analysis environment for representing and exploring alternative architectural design choices. DuSE supports the definition of design dimensions capturing prominent design decisions, the use of OCL validation rules to check for invalid solutions, the definition of metrics to evaluate resulting architecture quality attributes, and a search-based optimization algorithm to elicit a set of locally Pareto-optimal candidate architectures. We have also described SA:DuSE – the specific DuSE instance created to tackle design decisions inherent in the domain of self-adaptive systems. We have presented some details about our supporting tool, DuSE:MT, and prelimary results from a case study in a cloud-based media encoding service.

We are currently performing further experiments aimed at evaluating the optimality of the generated solutions, analyzing the statistic significance of resulting Pareto-fronts, and investigating the efficiency of the proposed metrics to elicit major trade-offs when designing self-adaptive systems. Case studies in other domains which require self-adaptation capabilities, such as middleware platforms and embedded systems, are also currently been investigated. The use of quality indicators

for evaluating the resulting Pareto-front and the definition of domain-specific tactics which lead to faster solution convergence are also under investigation.

Future work include the use of alternative approaches for multi-objective optimization – such as Ant Colony Optimization and Multi-Objective Simulated Annealing [22], the definition of annotated design space navigation traces to document design rationale and support for sharing models and design spaces over a network.

ACKNOWLEDGMENT

This research is partially supported by grant 006/2012/PRPGI from the IFBa's Research and Innovation Supporting Program.

REFERENCES

[1] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 14:1–14:42, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1516533.1516538

[2] R. de Lemos et al, "Software Engineering for Self-Adaptive Systems: A second Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Dagstuhl Seminar Proceedings, R. de Lemos, H. Giese, H. Müller, and M. Shaw, Eds., no. 10431. Dagstuhl, Germany.: Schloss Dagstuhl. Leibniz-Zentrum fuer Informatik, Germany. Full citation: http://dx.doi.org/10.1007/978-3-642-02161-9_1, 2011. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2011/3156

[3] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Software engineering for self-adaptive systems," in *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Engineering Self-Adaptive Systems through Feedback Loops, pp. 48–70. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02161-9_3

[4] R. Hebig, H. Giese, and B. Becker, "Making control loops explicit when architecting self-adaptive systems," in *Proceedings of the 2nd International Workshop on Self-Organizing Architectures*, ser. SOAR '10. New York, NY, USA: ACM, 2010, pp. 21–28. [Online]. Available: http://doi.acm.org/10.1145/1809036.1809042

[5] H. Müller, M. Pezzè, and M. Shaw, "Visibility of control in adaptive systems," in *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, ser. ULSSIS '08. New York, NY, USA: ACM, 2008, pp. 23–26. [Online]. Available: http://doi.acm.org/10.1145/1370700.1370707

[6] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A systematic survey on the design of self-adaptive software systems using control engineering approaches," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, june 2012, pp. 33 –42.

[7] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[8] M. Shaw, "Research toward an engineering discipline for software," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 337–342. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882431

[9] S. Gregor, "The nature of theory in information systems," *MIS (Management Information Systems Research Center) Quarterly*, vol. 30, no. 3, pp. 611–642, Sep. 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=2017296.2017300

[10] D. Jones and S. Gregor, "The anatomy of a design theory," *Journal of the Association for Information Systems*, vol. 8, no. 5, 2008.

[11] D. I. K. Sjøberg, T. Dybå, B. C. D. Anda, and J. E. Hannay, *Building Theories in Software Engineering*. Springer-Verlag London, 2008, ch. 12, pp. 312–336.

[12] M. A. Babar, T. Dingsyr, P. Lago, and H. van Vliet, *Software Architecture Knowledge Management: Theory and Practice*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[13] J. Branke, K. Deb, K. Miettinen, and R. Slowinski, Eds., *Multiobjective Optimization, Interactive and Evolutionary Approaches [outcome of Dagstuhl seminars]*., ser. Lecture Notes in Computer Science, vol. 5252. Springer, 2008.

[14] E. Burke and G. Kendall, Eds., *Search Methodologies: Introductory Tutorials in Optimisation and Decision Support Techniques*. Springer, 2005. [Online]. Available: http://www.springerlink.com/content/978-0-387-23460-1

[15] F. P. Brooks, *The Design of Design: Essays from a Computer Scientist*, 1st ed. Addison-Wesley Professional, 2010.

[16] M. Shaw, "The role of design spaces," *IEEE Software*, vol. 29, no. 1, pp. 46–50, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1109/MS.2011.121

[17] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, Dec. 2012. [Online]. Available: http://doi.acm.org/10.1145/2379776.2379787

[18] O. RäIhä, "Survey: A survey on search-based software design," *Comput. Sci. Rev.*, vol. 4, no. 4, pp. 203–249, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.cosrev.2010.06.001

[19] S. S. Andrade and R. J. d. A. Macêdo, "Architectural design spaces for feedback control concerns in self-adaptive systems," in *Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering*, ser. SEKE 2013. New York, NY, USA: ACM, 2013.

[20] A. O'Dwyer, *Handbook of Pi And Pid Controller Tuning Rules*, 2nd ed. Imperial College Press, Feb. 2006. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1860946224

[21] K. J. Astrom and B. Wittenmark, *Adaptive Control*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994.

[22] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *Trans. Evol. Comp*, vol. 6, no. 2, pp. 182–197, Apr. 2002. [Online]. Available: http://dx.doi.org/10.1109/4235.996017

[23] D. Weyns, S. Malek, and J. Andersson, "Forms: Unifying reference model for formal specification of distributed self-adaptive systems," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 7, no. 1, pp. 8:1–8:61, May 2012. [Online]. Available: http://doi.acm.org/10.1145/2168260.2168268

[24] T. Vogel and H. Giese, "A language for feedback loops in self-adaptive systems: Executable runtime megamodels," in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012)*. IEEE Computer Society, 6 2012, pp. 129–138.

[25] F. Křikava, P. Collet, and R. B. France, "Actor-based runtime model of adaptable feedback control loops," in *Proceedings of the 7th Workshop on Models@run.time*, ser. MRT '12. New York, NY, USA: ACM, 2012, pp. 39–44. [Online]. Available: http://doi.acm.org/10.1145/2422518.2422525

[26] N. Esfahani, A. Elkhodary, and S. Malek, "A learning-based framework for engineering feature-oriented self-adaptive software systems," *IEEE Transactions on Software Engineering (to appear)*, 2013.

[27] S.-W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based self-adaptation in the presence of multiple objectives," in *Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems*, ser. SEAMS '06. New York, NY, USA: ACM, 2006, pp. 2–8. [Online]. Available: http://doi.acm.org/10.1145/1137677.1137679

[28] A. Koziolek, "Automated improvement of software architecture models for performance and other quality attributes," Ph.D. dissertation, Institut für Programmstrukturen und Datenorganisation (IPD), Karlsruher Institut für Technologie, Karlsruhe, Germany, 2011. [Online]. Available: http://sdqweb.ipd.kit.edu/publications/pdfs/koziolek2011g.pdf

[29] T. Saxena and G. Karsai, "A meta-framework for design space exploration," in *Engineering of Computer Based Systems (ECBS), 2011 18th IEEE International Conference and Workshops on*, 2011, pp. 71–80.