A Design Space for Self-Adaptive Systems

Yuriy Brun[1], Ron Desmarais[2], Kurt Geihs[3], Marin Litoiu[4],
Antonia Lopes[5], Mary Shaw[6], and Michael Smit[4]

[1] Computer Science & Engineering, University of Washington, Seattle WA, USA
brun@cs.washington.edu
[2] University of Victoria, Vancouver, Canada
rd@uvic.ca
[3] EECS Department, University of Kassel, Kassel, Germany
geihs@uni-kassel.de
[4] York University, Toronto ON, Canada
mlitoiu,msmit@yorku.ca
[5] Department of Informatics, University of Lisbon, Lisboa, Portugal
mal@di.fc.ul.pt
[6] Institute for Software Research, Carnegie Mellon University, Pittsburgh PA, USA
mary.shaw@cs.cmu.edu

Abstract. Self-adaptive systems research is expanding as systems professionals recognize the importance of automation for managing the growing complexity, scale, and scope of software systems. The current approach to designing such systems is ad hoc, varied, and fractured, often resulting in systems with parts of multiple, sometimes poorly compatible designs. In addition to the challenges inherent to all software, this makes evaluating, understanding, comparing, maintaining, and even using such systems more difficult. This paper discusses the importance of systematic design and identifies the dimensions of the self-adaptive system design space. It identifies key design decisions, questions, and possible answers relevant to the design space, and organizes these into five clusters: observation, representation, control, identification, and enacting adaptation. This characterization can serve as a standard lexicon, that, in turn, can aid in describing and evaluating the behavior of existing and new self-adaptive systems. The paper also outlines the future challenges for improving the design of self-adaptive systems.

Keywords: adaptive, self-adaptive, design, architecture

1 Introduction

Designing a self-adaptive software system involves making design decisions about how the system will observe its environment and choose and enact adaptation mechanisms. While most research on self-adaptive systems deals with some subset of these decisions, to our knowledge, there has been neither a systematic study of the design space nor an enumeration of the decisions the developer should address. Developers draw from their own backgrounds to make decisions about self-adaptive system design. At worst, the design is predicated on modifying the existing system until it appears to work.

A *design space* is a set of decisions about an artifact, together with the choices for these decisions. A design space serves as a general guide for a class of applications, enumerating decisions and alternatives to provide a common vocabulary for describing, understanding, and comparing systems. A designer seeking to solve a problem may be guided by the design space, using it to systematically identify required decisions, their alternatives, and their interactions.

Intuitively, a design space is a k-dimensional Cartesian space in which design decisions are the k dimensions, possible alternatives are values on those dimensions, and complete designs are points in the space. In practice, most interesting design spaces are too rich to represent in their entirety, so their representations capture only the principal decisions as dimensions. Also in practice, the design dimensions are neither independent nor orthogonal, so choosing an alternative for one dimension may affect other dimensions and preclude or make irrelevant other decisions [3, 16, 20]. Creating a design space leads to creating rules, guidelines, and best practices that identify good and bad combinations of choices.

In this paper, we outline a design space for self-adaptive systems with five principal clusters of dimensions (design decisions). The clusters are observation, representation, control, identification, and enacting adaptation. Each cluster provides additional structure in the form of questions a designer should consider. These questions are not necessarily novel themselves; indeed, some of them are fundamental to self-adaptive systems. Rather, we hope our enumeration and organization will help formalize and advance the understanding of self-adaptive system design as a whole. The formulation as a design space is intended to explicitly help design systems. We do not present this as a final and definitive design space; further work on expanding and refining this design space is necessary and appropriate.

The remainder of this paper is organized as follows. Section 2 identifies related work on characterizations and models for self-adaptive systems, and discusses how presenting a design space is a novel contribution. Section 3 presents (1) a basic architecture and terminology and (2) an example system. Section 4 describes the design space and Section 5 places the example system in that space. Section 6 outlines future challenges for defining the self-adaptive system design space. Finally, Section 7 summarizes our contributions.

2 Related Work

Examples of self-adaptive systems include autonomic computing systems and self-managing software systems. This broad term includes self-configuring, self-healing, self-adapting, and self-protecting. IBM has identified a four-stage cycle for autonomic computing, called MAPE-K: Monitor, Analyze, Plan, and Execute, with a Knowledge Base common to all components (see Figure 1) [12, 14]. Others have referred to this cycle as the collect / analyze / decide / act cycle [7]. The architectural model that employs such a cycle implements a feedback loop with the Controller comprising of the four stages of MAPE-K. This Controller uses measured output from sensors monitoring the managed system to choose control signals to send to the managed system. The Monitor stage uses sensors to measure key attributes, usually related to the current performance

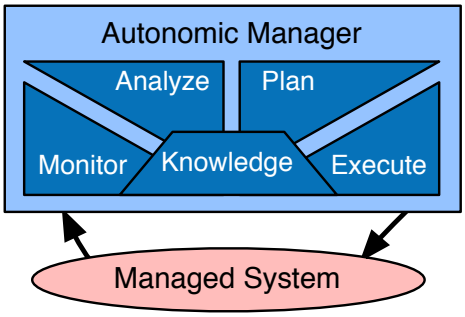

Fig. 1. The MAPE-K model (based on [14])

and load of the system. The Analyze stage identifies any metrics not within tolerances (or violating some type of defined rule) and attempts to identify the cause or source of the problem. In the Plan stage, the system reacts to the fault (or generally, the results of the Analyze stage) by identifying a set of actions that remedy the situation. These actions are implemented in the Execute stage via actuators that act on the managed systems. All the stages make use of a Knowledge Base.

Several partial approaches to identifying and representing design spaces for self-adaptive software systems are available. These existing approaches have their roots in both software engineering (e.g., [7, 15]) and control theory (e.g., [4, 22]).

Andersson et al. [1] defined a set of modeling dimensions for self-adaptive software systems. The dimensions were organized into four groups, including the self-adaptive goals of the system, the causes or triggers of self-adaptation, the mechanisms used to adapt, and the effects of those mechanisms on the system. Their dimensions have some overlap with our dimensions; they and we both explicitly encourage the addition of more dimensions. Though parts of our work extend theirs, and we consider their work important and relevant, we offer a redevelopment with important differences that are in part related with the differences between modeling and design:

First, their dimensions are appropriate for classifying an existing adaptive system, while ours are more appropriate for designing a system based on a set of requirements. Their categories could be said to be driven by observation; that is, the inputs, the causes, and the effects. We make explicit the design decisions regarding how we observe and represent the inputs, how these inputs are translated into adaptation triggers, and what the capabilities and limits of our adaptation mechanisms are.

Second, our work aims at a higher level of abstraction. Some of their dimensions would be implementation decisions in our design space. For instance, whether one or several components are responsible for adaptation is a design dimension in their work.

Third, while there is some overlap among the categories of dimensions, each makes different exclusion and inclusion decisions. They view the adaptation goal as one of the major categories (having 5 dimensions). We assume the goals (called adaptation targets) are dynamic, multiple, and specified at run-time not design time (and therefore not explicitly included in the design space). However, our design space is still applicable to

static, singular adaptation targets. We identify several key dimensions not included in their work, in particular those related to modeling, representation, and control loops. For example, though perhaps not essential to a working implementation, an explicit conceptual understanding of how the control loops are orchestrated improves the developers' understanding. We include assessing what is possible: what can be observed, what can be adapted, whether the managed system provides appropriate mechanisms to achieve the desired control, etc.

Kramer and Magee [15] outline three tiers of decisions the developer must make — ones that pertain to goal management, change management, and component control. Finally, Brun et al. [4] discuss the importance of making the self-adaptation control loops explicit during the development process and outline several types of control loops that can lead to self-adaptation.

Villegas et al. [22] describe a characterization model for approaches to self-adaptive systems that include a survey of existing self-adaptive systems. They systematically identify what these systems use for each element of self-adaptation (these elements are described here in Section 3). Their results support our claim that current systems are often designed using an ad-hoc approach. This post-hoc analysis of past implementations is interesting and useful; their focus is on evaluating and characterizing self-adaptive systems post-implementation, rather than guiding design decisions at design time.

Taxonomies for self-adaptive systems have also been proposed. For example, Brake et al. [2] and Ghanbari et al. [10] introduce and discuss a taxonomy for performance monitoring of self-adaptive systems together with a method for discovering parameters in source code. Checiu et al. [6] formally defined controllability and observability for web services and showed that controllability can be preserved in composition.

3 Self-adaptive Systems

This section describes (1) a basic architecture and terminology for self-adaptive systems used as a working definition throughout the paper, and (2) a web application that illustrates this terminology and will serve as a running example throughout the remainder of this paper.

We separate self-adaptive systems into two elements: the Managed System, which is responsible for the system's main functionality, and the Adaptation System, which is responsible for altering the Managed System as appropriate. Figure 2 shows these two elements linked by the flow of system indicators and adaptations. The elements inherent to the managed system (that is, the things that would exist even if it were not adaptively managed) such as the inputs and the environment are captured and used by the Adaptation System. The Adaptation System produces adaptations that impact the Managed System.

Example. To illustrate these concepts, consider the following example which will be used throughout the paper to illustrate the design space. This generic example is representative of a large class of self-adaptive systems.

Consider a web application running on one or more web servers. This application must support an arbitrary number of users with a response time below a predefined threshold. The application accesses a database, but this is only a potential bottleneck

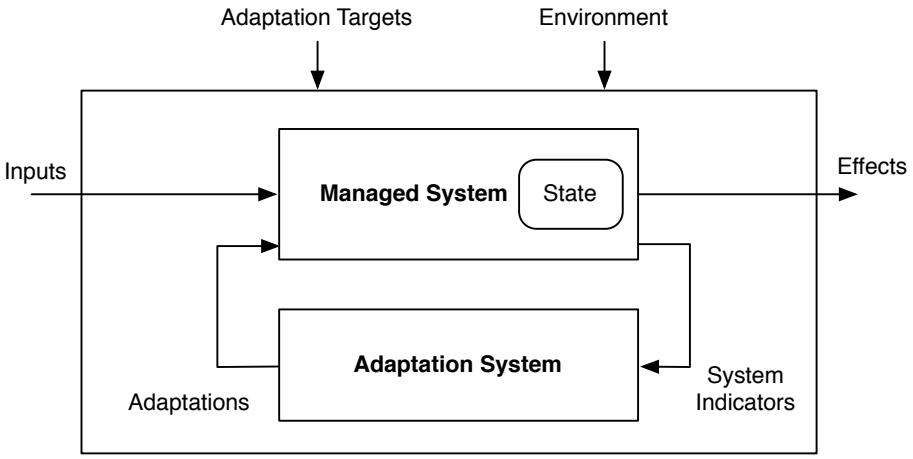

Fig. 2. Elements of a self-adaptive system. The arrows indicate information flow.

when certain intensive features are used, like the recommendation engine. The application must also be available for a predefined minimum percentage of time, except for defined maintenance windows. The operational and environmental conditions (e.g., the number of users) change during execution. Elements of the application or the hardware server may fail at any time. The web application proper (Managed System) is augmented with an adaptation driver (Adaptation System) that will monitor the application and select and apply the necessary changes at runtime.

Definitions. The information used by the Managed and Adaptation System is described in more detail below.

- *Adaptation Targets* define what the self-adaptation should achieve. Adaptation targets are not necessarily end user requirements; they could be design goals derived by determining what is required to meet end user requirements in an optimal manner. For instance, in our example, the end user requirement *response time below a threshold* could result in an adaptation target outlining how to achieve this requirement with minimum resources. These targets can be expressed in different ways depending on the domain and the design decisions.
- *Effects* are what Managed Systems produce per their functional requirements; however, included are *system indicators* (of functional or non-functional properties) which can be evaluated to determine how compliant the managed software is with the adaptation targets, and are part of the input to the Adaptation System. In our example, the *response time* of the web application would constitute an effect parameter. The evaluation process may be more complex than comparing a momentary value to an established threshold. Again in the example, the *availability over time* effect requires capturing availability information over time and calculating an average over a sliding window of time.
- *Adaptations* are the actions taken by the Adaptation System. One type of adaptation is *parametric adaptation*, the adjusting of tuneable parameters to attain the

desired behavior from the Managed System. The correct tuning for these parameters is computed by the Adaptation System by taking into account the System Indicators, Environment and State. In the web application, a tuneable parameter might be the *number of threads* used to service user requests. By adjusting the number of threads, the State is changed (specifically the thread queue length state) and the Effect and System Indicators are affected (response time). Another type is *architectural adaptation* (or composition adaptation), for instance exchanging one component for another. In our example system, depending on how the change is effected, changing the type of recommendation engine would be an example of exchanging one component for another.

- *State* is the representation of the internal state of the Managed System, and is comprised of a collection of state parameters that characterize or model the Managed System. The state of the Managed System is affected by the Inputs and Environment streams, and plays a role in determining the Effects and System Indicators. In our web application example, the *length of the thread queue* serving user requests could be a State parameter. Other parameters could be identified based on their ability to accurately characterize or model what is happening inside the Managed Application. The State used in an adaptive system is highly dependent on which System Indicators can be measured and with what accuracy, a challenge discussed further in the Observation cluster (Section 4.1).
- *Environment (or perturbation)* indicators are external to the self-adaptive software and the Managed System, but have a direct or indirect influence on the State, Effects, and System Indicators. For our example, the *workload* (e.g., the number of users accessing the application running on the web server or the frequency of their requests) is an environment indicator. These indicators typically cannot be controlled (directly or indirectly).

4 Dimensions of the Design Space

We describe the design space for self-adaptive systems using various *dimensions*, each defined by a design question that admits more than one possible answer (the design decision). The dimensions are from our own experience designing self-adaptive systems, and can and should be extended with the experience of others. To manage the complexity, we group dimensions into *dimension clusters*, such that each cluster represents a particular category of concern. The clusters, shown in large font in Figure 3, represent the design space. There is natural overlap between the clusters (indicated by dimensions affecting multiple clusters) and dependence between some dimensions (indicated by similarity of color). The clustering helps manage the complexity of the design space, but it is also possible to consider the dimensions independently of their clusters.

The *Observation* cluster includes questions related to what is monitored by the Adaptation System, when and how often monitoring occurs, and how states are determined based on the monitored data. The *Representation* cluster is concerned with the runtime representation of adaptation targets, inputs, effects, system indicators, and states. The *Control* cluster is concerned with the mechanisms through which a solution is enacted. While these three dimensions are somewhat independent, they have more

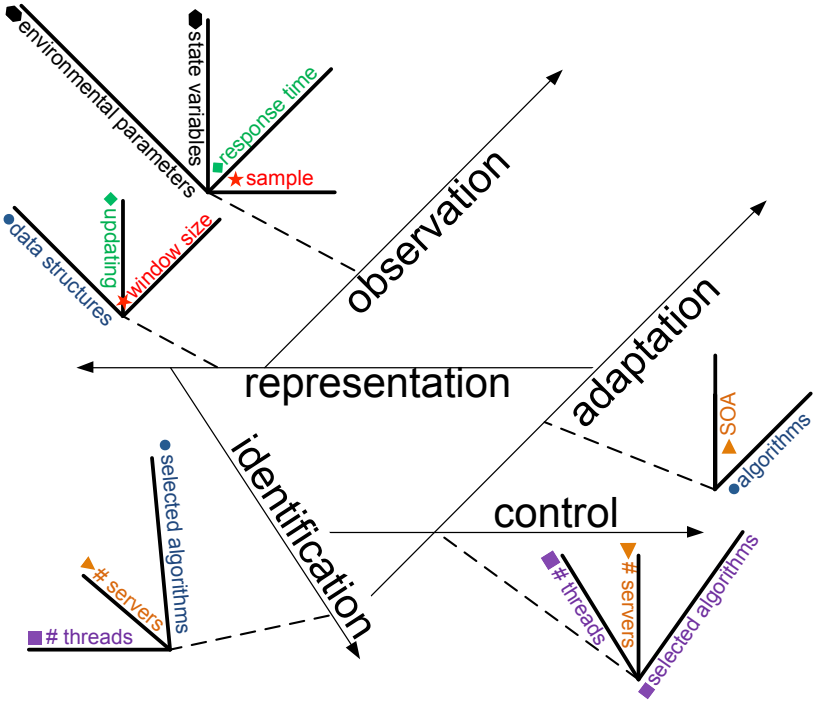

Fig. 3. The dimensions of the self-adaptive system design space can be grouped into five clusters, shown here with a sampling of properties for our example system. Intersections among the dimension lines indicate potential overlap. Sample properties with bulleted text in the same colour represent dependencies. For example, the *window size* chosen in representation depends on the *sample* size available in observation.

significant overlap with the final two. The *Identification* cluster is concerned with the possible solution instances that can achieve the adaptation targets. Finally, the cluster of *Enacting Adaptation* concerns how adaptation can be achieved.

Table 4 details the clusters and each cluster's dimensions. For each cluster, we name the key dimensions (in the form of questions) and by way of example suggest answers to these questions. When using the design space as a guide for software design, it is important that the answers be more specific and implementation-focused. For instance, for the question "What triggers observation of metric m?" a general answer is "periodically, on a timer," whereas a more-specific answer is "every k seconds, for $k \in [1, 2, \ldots, 1000]$ sec." The level of appropriate abstraction depends on the design space's use. Section 5 steps through the dimension clusters answering design questions for the sample application described in Section 3.

While we hope our enumeration will help formalize and advance the understanding for self-adaptive system design, it is not intended to be complete and further work on expanding and refining this design space is necessary and appropriate.

Observation – what can be observed? – what information will the system observe? – what information about the external environment and about the system itself will need to be measured and represented internally? – how will the system determine that information? – what triggers observation? – what triggers adaptation? – how is uncertainty in the observations handled?
Representation – what information is made available to the components of the self-adaptive system? – how is this information represented? – when and how is the information updated?
Control – does the system provide enough operations/parameters to achieve the desired control? – how does the system decide what and how much to change to modify its behavior? – how will the control loops be orchestrated? – what to adapt? – how to adapt? – when to adapt?
Identification – what are possible solutions for a given set of adaptation targets? – which solutions are or will be supported? – what are relevant domain assumptions and contexts for each solution? – what are the required observed and control parameters for each solution?
Enacting Adaptation – is the adaptation mechanism represented explicitly or implicitly in the architecture? – how is adaptability supported? – how will failures of the adaptation mechanism be handled? – what is the cause of adaptation?

Fig. 4. Dimensions of the design space and their primary questions, organized into clusters

4.1 Observation

The observation cluster includes dimensions covering design decisions regarding what information is observed by the Adaptation System. Generally, the dimensions in this cluster are related to the question "How do we gather information about the Managed System (System Indicators and States) and its Environment?" This cluster is related to the representation cluster: observations gathered with no mechanism for internal representation are superfluous, and internal representations not updated by observations are potentially serious problems. (Although we acknowledge that some reactive systems may act on observations directly, without internalizing them in a representation.)

At a conceptual level, the information gathered in a self-adaptive system is intended to be a proxy measure for one or more higher-level constructs, such as workload, performance, or safety. The designer chooses system and environmental properties that they expect to be good predictors of these constructs. However, the ability to measure these properties is often limited. For instance, to decrease the overhead of monitoring, periodic samples are taken, and intermediate values are inferred. If physical elements are involved (e.g., CPU temperature, server room temperature, voltage, etc.), there may be limits on where a sensor can be placed, and on the accuracy of that sensor. The designer must be aware of what value the self-adaptive system will be using. If not precisely the CPU temperature, is it the temperature at some point near the CPU where a sensor can be installed? If so, can the value returned by the sensor be trusted? Or should it instead be a value derived from the sensor but corrected by calibration and averaging? If so, what inaccuracies are introduced by the time lag waiting for these measurements?

We wrap up these questions into a single question: "what *can* be observed?" For any system property of interest (perhaps because that's a target, perhaps because it's used in a model), the designer must understand if it is possible to observe directly, if it must be indirectly estimated by inference or aggregation, etc. In short, the designer must distinguish between "what the system wants to measure" and "what the system can measure and infer from measurements." This makes it explicit that the self-adaptive system relies on the ability of the observations to actually estimate the desired system property, and may even impose proof obligations on the designer to show that what the system is capable of measuring is an acceptable proxy for the higher-level construct that is conceptually important.

A key design decision about self-adaptive systems is "what information *will* the system observe?" In particular, "what information about the external environment and about the system itself will need to be measured and represented internally?" The self-adaptive system may require awareness of the context within which the managed system is operating (e.g., [9]). Answering these questions requires an understanding of how the information will be used, which may require other design decisions to be made first.

Given the set of information the system observes, another important design decision is "how will the system determine that information?" The system could make direct measurements with sensors, use measurements provided by an already instrumented system, infer information from a proxy, extrapolate based on earlier measurements, aggregate knowledge, etc. Some of the measurements can be made implicitly, e.g., by inferring them from the state of the system or success or failure of an action. Others will require aggregation or reasoning over time or events, or be in some way combined with previous measurements. The answer may be different for each metric being observed.

Knowing what to observe and how it is observed, the next dimension is when to observe. There are actually two timing questions: "what triggers observation?" and "what triggers adaptation?" The system could be continuously observing or observation could be triggered by an external event, a timer, an inference from a previous observation, deviation or error from expected behavior, etc. Thus, the observation can happen at a fixed delay, on-demand, or employ a best-effort strategy. The same decisions relate to when the adaptation triggers. Again, this question may need to be answered differently for each item in the set of information being observed.

A final set of decisions dealing with observation is "how is uncertainty in the observations handled?" Answers could include filtering, smoothing, and redundancy, or perhaps the system might not have a specific strategy to deal with noise and uncertainty.

4.2 Representation

The representation cluster is concerned with design decisions regarding how the problem and system are represented at runtime. To enable mechanisms of adaptation, key information about the problem and system may have to be made available at runtime. As mentioned, this cluster is closely related to the observation cluster, and has ties to the remaining clusters that make use of this representation.

A key decision in this cluster is "what information is made available to the components of the self-adaptive system?" Answers include different aspects of the adaptation targets, existing domain knowledge, and observations of the environment and Managed System that are found necessary and sufficient for self-adaptive system to operate.

Another design decision in this cluster relates to choices of internal representation. "How is this information represented?" is meant to guide the designer to the representation that best matches the adaptation targets and the nature of the problem. Choices include explicit representations such as graph models, formulae, bounds, objective functions, etc., or implicit representations in the code.

Since some of the information in the representation is dynamic — primarily observations of the environment and the Managed System, but potentially also the adaptation targets or the knowledge model — it is necessary to have an approach to update the representation: "when and how is the information updated?" This dimension prompts a decision regarding when updates will occur (a fixed delay, on-demand, best effort, etc.), and how these updates occur (push? pull? with notification to concerned components?). Of course, static information will not be updated; another important design decision is what information is static.

4.3 Control

This cluster is concerned with the mechanisms whereby system execution changes the Managed System in some way to bring it in line with the adaptation targets.

Perhaps the most important question regarding this cluster is "does the system provide enough operations or parameters to achieve the desired control?" The designer has the obligation to show that the choices made in the Identification cluster are sufficient to achieve the desired adaptation targets.

One dimension in this cluster is "how does the system decide what behavior to modify and by how much?" Deciding "what" will change depends on knowing what adaptations (control inputs, architectural changes, deployment changes, etc.) are available, and then identifying those that will modify the system appropriately. The amount of the change can be a predefined constant value, or can be proportional to, or a function of the deviation from the desired behavior, or of another factor. In some cases, the change is a sum of three terms, a control technique known as *PID*: a component proportional with the control error (*P*, roughly the error at the time), a component proportional with the derivative of the error (*D*, roughly the rate of change of the error)

and another proportional with the integral of the error (I, roughly the accumulated error over time). In addition to the PID controller, other control-systems analogies can also be used. In situations with the target of modifying along several requirements, optimizing a utility function may be appropriate. Complex modifications may require planning and re-planning.

Adaptation is realized through feedback loops and complex software systems require multiple loops to adapt [4]. A key decision is "how will the control loops be orchestrated?" Possible answers depend on the structure of the system and the complexity of the adaptation targets. Composition patterns include series, parallel, multi-level (hierarchical), nested, and independent loops. The control loops may not be explicit in all self-adaptive systems, but some understanding of the nature of the loops (feedback, feedforward, etc.) is important to guide the design.

Control loops need to be further augmented with actuation decisions. The design questions "what to adapt?" can have several answers. We can adapt parameters, representation, resource allocation, component and connector selection, component and connector synthesis, sensor and actuator augmentation, component deployment, etc. Once the actuators are selected, the next question becomes "how to adapt?" This decision provides concrete actuation methods, including abort, modify data, call procedure, start new process, etc. Like observation and representation, there is a time dimension in control. "When to adapt?" can lead to choices such as: immediately, on a time scale, with fixed delay, continuous change, lazy, on-demand, best effort, etc.

4.4 Identification

The identification cluster defines the possible solutions a self-adaptive system instance can assume when it adapts. The solutions can cover changes in system structure, in its behavior, or in a combination of both.

The first dimension in this cluster identifies adaptation solutions: "what are possible solutions for a given set of adaptation targets?" Finding a discrete set of possible structural changes, states, parameter values, etc. is the main concern. Changes can include parametric adaptations where some tuneable parameter is adjusted, architectural (compositional) adaptations where software components or their connections are replaced or updated, and deployment adaptations where the deployment of the adaptation system is changed (e.g., moving it to the public cloud). Not all solutions will be supported at runtime. Answering the question "which solutions are or will be supported?" narrows the list to a subset that will be available at runtime.

Another set of dimensions identifies runtime contexts and feedback loop parameters for a particular set of solutions. The "what are relevant domain assumptions and contexts for each solution?" design question provides the runtime context for the available self-adaptive solutions. "What are the required observed and control parameters for each solution?" identifies prerequisites; this information should be taken into consideration in the observation and representation clusters. It also provides the necessary information to enable the changes in feedback loop components when switching from one solution to another.

4.5 Enacting Adaptation

The Enacting Adaption cluster includes the adaptation mechanisms, how they are triggered, how they are supported, and how failure is handled; it is the closest of the clusters to the implementation solution.

A first question to ask is "is the adaptation mechanism represented explicitly or implicitly in the architecture?" Some feedback loops can be represented explicitly in the architecture whereas others are intrinsic to the system functionality and therefore represented implicitly. A self-adaptive system can also be a hybrid of explicit and implicit loops.

The answers to the question "how is adaptability supported?" might depend on issues not related to adaptivity itself but on other system goals, such as maintainability and reliability. This could be answered in terms of the architecture of the Managed System (a plugin architecture, web services style, etc.) or in terms of the adaptations available on the Managed System. Closely related to the previous design question is "how will failures of the adaptation mechanism be handled?" e.g., try again, adopt another mechanism, etc.

Design decisions in all clusters will be affected by what causes adaptation, so it is important to answer the question of "what is the cause of adaptation?" Answers include: non-functional requirements (these often use control-theoretic concepts and relate to response time, throughput, energy, consumption, reliability, fault tolerance), behavior, prevention of undesirable events, maintaining state, dealing with anticipated change, dealing with unanticipated change, etc.

5 Using the Design Space

The design space is intended to be useful for tasks such as describing the behavior of existing self-adaptive systems using a shared standard lexicon, characterizing or creating a taxonomy for self-adaptive systems, evaluating a self-adaptive system using the design space as a checklist, comparing several systems or designs, and designing a new self-adaptive system. The primary intended use for the design space is to serve as a guide when designing a self-adaptive system. The questions of each dimension either can be answered throughout the design document or can be explicitly separated into their own chapter.

To illustrate this use of the design space, we show here the design decisions for our sample managed web-server system (Section 3), with the adaptation target of achieving response time within a given limit for 95% of requests, while controlling costs. Note we have not explicitly shown these design decisions are the right ones; that is, we show what a point in the design space looks like, but do not make a claim that it is the correct point (or one of the several possible correct points).

5.1 Observation

The Adaptation System will observe the effect Response Time; the internal state parameters Actual Number of Threads, Thread Queue Length, CPU Utilization; and the

environmental indicators of workload Request Arrival Rate and Number of Users. The Adaptation System also observes the state of the available control inputs — number of threads, number of servers, and the algorithm used by the Recommendation Engine.

The effect and environmental indicators are captured at the application boundary (measuring incoming requests and outgoing responses); the state parameters and inputs are captured using sensors built into the application itself. Observations at the application boundary are made after every k requests and reported in the aggregate (median, mean, standard deviation) over those k requests, with a configurable k (default is $k = 1000$). The state parameters are observed every j seconds, with a configurable j (default $j = 180$). Control inputs are observed, and can be changed every i minutes, with i configurable (default $i = 10$). Uncertainty in the observations is not handled by the observation and monitoring components, though we note that high variance in the data can be measured by observing its standard deviation.

5.2 Representation

The information observed is available, both current and historic for a configurable length of time. Other than the information discussed in the Observation cluster above, the system also keeps track of

- 95^{th} percentile response time over various sliding windows of time (contract period, last day, last hour) to measure compliance with the adaptation target,
- the current cost of the configuration (computed based on control inputs and current architecture),
- past configuration and workload pairs that have met the adaptation target, and
- the current target required by the adaptation target.

The software components can access the observed information though internal data structures. Known-good configuration and workload pairs are represented in a machine-readable knowledge base, translated into a set of logical rules that can be used to produce recommendations or to power an expert system. (For the purpose of this example, we leave aside the details of the expert system design and treat it as an oracle loosely coupled to the Adaptation System.) The adaptation targets are expressed as objective functions.

The adaptation targets must be updated manually (not automatically). The rest of the represented information is dynamic and will be updated as follows:

- Rules are added to the knowledge base without notification to any components. The intent for is for the rules to considered in the next recommendation produced by the knowledge base.
- The software data structures used to store and access observed information ensure that any request for information will always return the most up to date information available. Such requests will block if there is a pending or overdue attempt to update. The data structure is updated as the information is observed (see the Observation cluster above).

5.3 Control

The controls available to our system are: adding and removing a thread, and adding and removing a server. The available architectural change is swapping out the recommendation engine component. The number of threads that can be added is limited by the resources available on the currently deployed servers. The number of servers is theoretically not bounded. Resources scale non-linearly: adding a thread for which there is enough capacity is quite inexpensive, but adding a whole server is more expensive. (We assume a utility computing model, such as the cloud, where additional scaling — adding a rack or adding data center capacity — is not required.) The Recommendation algorithm can be either highly personalized (database intensive) or generic (with low resource usage). Our example does not allow for database replication, so we do not consider the complexity of added revenue from a personalized recommendation engine. Threads can be removed one at a time and are added proportionally to how overloaded existing threads are. Servers are added and removed one at a time. The decision to change is based on the optimization of the target adaptation objective function and the recommendation of the expert system and knowledge base.

For this simple example, there is one primary feedback loop that manages response time. There are simple, nested feedback loops that identify corrective actions based on the current state of the control inputs. If we included other adaptation targets (e.g., managing uptime), there would be parallel feedback loops.

The actuator for adding a thread launches a lightweight process and adds an entry to the load balancer. Removing a thread involves identifying the oldest lightweight process, removing its entry from the load balancer, stopping it safely, and monitoring to ensure the stop command succeeds (in necessary, the process can be killed). The actuator for adding a server involves a utility-computing command to deploy a new server from a given machine image and to trigger the thread actuator to add new threads. Server removal requires safe termination of the threads and utility-computing commands. Changing the recommendation engine algorithm involves making a call to the web application API, then updating the internal representation of the Managed System to note the reduced resource usage that will enable more threads per server. All actuations of adaptations occur immediately.

5.4 Identification

Adaptation solutions define architectural changes and ways of influencing the state and effect variables through changes in control inputs. Alternative solutions can be explicitly represented in the design or implicitly represented in the algorithms and functions.

In the web-server example, changes will take place when:

- the expert system makes a recommendation, and
- the feedback loop identifies that additional resources are required, based on observations of the state of the application and the environment.

The self-adaptive solutions supported at runtime are combinations of:

A) change to the low-cost Recommendation Engine,

B) change to high-cost Recommendation Engine,
C) add threads to a server,
D) remove threads from existing servers, and
E) add a server.

The specific parameters and contexts for effecting each solution above, respectively:

A) When the database layer is the identified bottleneck (CPU utilization on the application server is low; thread queue is high).
B) When resource utilization is low and less than four application servers are in use.
C) When additional capacity is needed to ensure the adaptation target is met and when sufficient memory and CPU resources are available on a server but requests need to be processed faster.
D) When resource utilization is low and less than four application servers are in use.
E) When sufficient memory and CPU resources are not available on a server, requests need to be processed faster, and the utility cost of deploying a new server is less than the utility cost of violating the adaptation targets for the projected length of time the workload will remain high.

An important mechanism of evaluating adaptation solutions at runtime is the representation of the solution space through a quantitative model. Quantitative models are quality attribute-specific, and as such performance, reliability, security and availability have different models. For our example, we can choose from four major classes of performance models [18]. Queuing models can be implemented through simulation [21] or analytical models [17] and are easy to reconfigure at runtime for vertical and horizontal scaling of applications. Dynamic models such as regression models allow for a synthesis of the control using classic system control techniques. Policy models are a set of rules that capture the designer experience or can be discrete representations of more complex models. A Markov Decision Process model can be used to compute optimal reactions to state changes, combining observations on the system (for example, an observation of the system state) with assumptions about the rate of changes of state that are expected in the future.

5.5 Enacting Adaptation

The adaption mechanisms are built into the Managed System, and the observation and other elements of the feedback loop are explicitly designed and implemented directly in the code. The feedback loop is implicit. The Managed System is built with a service-oriented API and instrumentation to support being managed by the Adaptation System. When the provided adaptation mechanisms fail, the system will retry the adaptation at the next control interval. If the failure repeats, the system will attempt designated fallbacks (e.g., if adding a thread fails, the system will add a server). When the fallbacks are exhausted, the system will generate a notification that intervention is required and will retry the adaptations until it either unnecessary, works, or halted by the user.

The adaptation is caused by the expert system and through measurements on the Managed System.

6 Self-Adaptive System Design Space Challenges

While the design space described above helps formalize and advance the understanding of self-adaptive system design, it is not complete.

The main remaining challenge is to infuse a systematic understanding of the alternatives for adaptive control into the design process. Further, the trade-offs among the choices and quantitative and qualitative implications of the design decisions on the overall adaptation quality need to be investigated. The clusters of dimensions described here pose the questions that need to be answered by designers of self-adaptive systems. A complete solution, however, will also include the possible answers and a systematic way of evaluating the trade-offs of each choice (such as the SEI approach to architectural design [13]).

Another remaining challenge is to expand and refine the dimensions in each cluster. At the same time, since the dimensions are not independent, the dependencies among the dimensions and choices need to be understood. Such proper understanding can narrow the search through the design space, improve the efficiency of the design process, and reduce the design complexity. Validating the dimensions and their constraints against real-world examples can serve as the framework for describing alternatives and as a checklist to help designers avoid leaving out critical decisions.

This initial description has largely set aside the question of evaluating proposed designs. While the design space can be used as-is to ensure a design has considered the various dimensions, further work is required to help designers evaluate if their choices will lead to a system that achieves the desired control. Not all the points in the design space correspond to systems that will solve a given problem. How does the designer decide whether a given set of decisions (i.e., a point in the design space) suffices?

Bridging the gap from a generalized design space to an implementation is a complex challenge. The design space helps guide decisions but offers little guidance on the implementation of those decisions. Design patterns, architectural patterns, middleware, and frameworks can help bridge this gap. In particular, design patterns are solution templates designed to be reused [8]. The choices for each question in the design space could map to specific design patterns. Ramirez and Cheng [19] identify several design patterns specific to self-adaptive systems. Architectural patterns are to design patterns but target a higher level of abstraction, serving as blueprints that can be used and modified to suggest typical architectural solutions to common problems [5]. Middleware and frameworks and are more concrete and provide partial implementation, such as implementations of common tasks, common solutions, and common architectures for self-adaptive systems. As there are many proposed frameworks, architectures, and middleware for self-adaptive systems, selecting an appropriate approach based on choices made in the design space is an open, and non-trivial research problem.

For existing systems that need to be re-engineered to be self-adaptive, the design space may be even more constrained and complex than the space we have explored here. Extra dependencies and constraints may eliminate potential solutions in the design space. Tools can likely help understand this space, and the constraints.

Since complex self-adaptive systems will have more than one control loop, it is imperative that the common ways in which control loops interact be defined, along with patterns or common templates for handling these cases. For example, Hellerstein et

al. [11] describes two systems with multiple feedback loops. The first system is a self-tuning regulator, which adapts the controller's parameters, which, in turn, manage a plant. In this case, the plant (Managed System), does not need to be modified because the self-tuning regulator can determine the Managed System's internal state from its effect. The second system is a gain scheduler. For this system, the output of the Managed System is not sufficient to determine its state and the Managed System must allow itself to modified to provide the necessary scheduling variables. Thus, a remaining challenge is to better understand the control-loop-related questions that need to be answered in designing systems with multiple forms of feedback.

7 Conclusion

Before designing a self-adaptive software system, it is valuable to explore the design space of such systems and to understand the dimensions of the space. Further, the understanding of the design space can increase maintainability and interoperability of deployed systems and facilitate evaluation and enhancement of existing and proposed systems. In this paper, we have described the design space of self-adaptive systems in terms of five clusters of dimensions: observation, representation, control, identification, and enacting adaptation. Each dimension is a question a designer must answer. We have also proposed possible answers for each dimension question and discusses dimension dependencies. Finally, we have illustrated the design space with a self-adaptive web-server system.

This paper represents the next step in what we hope will be a confluence of several lines of research on the design of self-adaptive systems. The design space we have outlined here is not complete and not exhaustive. Further work defining and understanding the design space — primarily by adding additional key dimensions and possible answers to existing questions — will increase its utility. Understanding the trade-offs, dependencies, and best practices for the design space will further enrich the experience of self-adaptive system developers. Finally, establishing paths from the design space to the system implementation, via design and architectural patterns, frameworks, and middleware, will assist designers and developers.

We believe that understanding the design space of self-adaptive systems and following a more-systematic approach to such system design will ultimately improve the outcomes of self-adaptive systems research.

Acknowledgments We thank Gregor Engels, Jeff Magee, and John Mylopoulos for contributing ideas to the discussions that led to this paper.

References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems (LNCS 5525), pp. 27–47. Springer-Verlag, Berlin, Heidelberg (2009)

2. Brake, N., Cordy, J.R., Dancy, E., Litoiu, M., Popescu, V.: Automating discovery of software tuning parameters. In: Proceedings of the 3rd International Workshop on Software Engineering for Adaptive and Self-Managing Systems. pp. 65–72. Leipzig, Germany (2008)
3. Brooks Jr., F.P.: The Design of Design: Essays from a Computer Scientist. Addison-Wesley, New York (2010)
4. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems, vol. 5525, pp. 48–70. Springer-Verlag (2009)
5. Buschmann, F., Henney, K., Schmidt, D.: Pattern-oriented software architecture: On patterns and pattern languages, vol. 5. John Wiley & Sons Inc (2007)
6. Checiu, L., Solomon, B., Ionescu, D., Litoiu, M., Iszlai, G.: Observability and controllability of autonomic computing systems for composed web services. In: Proceedings of the 6th IEEE International Symposium on Applied Computational Intelligence and Informatics. pp. 269–274 (2011)
7. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1, 223–259 (December 2006)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc. (1995)
9. Geihs, K., Reichle, R., Wagner, M., Khan, M.U.: Modeling of Context-Aware Self-Adaptive Applications in Ubiquitous and Service-Oriented Environments, Software Engineering for Self-Adaptive Systems, vol. LNCS 5525, pp. 146–163. Springer-Verlag (2009)
10. Ghanbari, H., Litoiu, M.: Identifying implicitly declared self-tuning behavior through dynamic analysis. In: Proceedings of the 4th International Workshop on Software Engineering for Adaptive and Self-Managing Systems. pp. 48–57. Vancouver, BC, Canada (2009)
11. Hellerstein, J., Diao, Y., Parekh, S., Tilbury, D.: Feedback control of computing systems, pp. 378–384. Wiley Interscience (2004)
12. IBM: An architectural blueprint for autonomic computing. `http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf` (June 2006)
13. Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, J.: The architecture tradeoff analysis method. In: Proceedings of the 4th IEEE International Conference on Engineering of Complex Computer Systems. pp. 68–78 (1998)
14. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
15. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Future of Software Engineering. pp. 259–268 (2007)
16. Lane, T.G.: Studying software architecture through design spaces and rules. Tech. Rep. CMU/SEI-90-TR-18, Software Engineering Institute, Carnegie Mellon University (November 1990)
17. Litoiu, M.: Application performance evaluator and resource allocation tool (APERA). `http://www.alphaworks.ibm.com/tech/apera` (May 2003)
18. Litoiu, M., Woodside, M., Zheng, T.: Hierarchical model-based autonomic control of software systems. In: Proceedings of the Workshop on Design and Evolution of Autonomic Application Software. pp. 1–7. St. Louis, MO, USA (2005)
19. Ramirez, A.J., Cheng, B.H.C.: Design patterns for developing dynamically adaptive systems. In: Proceedings of the 5th International Workshop on Software Engineering for Adaptive and Self-Managing Systems. pp. 49–58. Cape Town, South Africa (2010)
20. Shaw, M.: The role of design spaces. IEEE Software, (special issue on Studying Professional Software Design) 29(1), 46–50 (2012)

21. Smit, M.: Supporting Quality of Service, Configuration, and Autonomic Reconfiguration using Services-Aware Simulation. Ph.D. thesis, University of Alberta (2011)
22. Villegas, N.M., Müller, H.A., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. In: Proceeding of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 80–89. Waikiki, Honolulu, HI, USA (2011)