Elicitation and Utilization of Utility Functions for the Self-assessment of Autonomic Applications

Paul de Grandis Drexel University paul.degrandis@drexel.edu

Abstract

We present a non-analytic approach to selfassessment for Autonomic Computing. Our approach leverages utility functions, at the level of an autonomic application, or even a single task or feature being exercised within that application. This paper describes the fundamental steps of our approach: instrumentation of the application; collection of exhaustive samples of runtime data about relevant quality attributes of the application, as well as characteristics of its runtime environment; elicitation of a utility function, through statistical correlation over the collected data points; and embedding of code corresponding to the equation of the elicited utility function within the runtime of the application, which enables online evaluation of utility values. To illustrate our elicitation method, as well as the prototype of our framework for instrumentation, monitoring, and utility function embedding/evaluation, we describe our experience with two different case studies, and discuss their results and implications.

1. Introduction

Autonomic software systems should be able to adapt to a wide range of conditions, including unforeseen and unexpected events that may occur as the system is operating. Such events may often cause the most critical in-the-field faults, since, by definition, they have not been accounted for, or discovered, during the various phases of the engineering process. It is therefore arguable that the significant additional cost and complexity incurred in designing and implementing autonomic mechanisms for a software system may at times be justifiable only in the case where they provide some level of protection from unpredictable conditions. However, self-adaptive facilities are often designed to cover only certain classes of *expected* or *likely* conditions, and *designing* Giuseppe Valetto Drexel University valetto@cs.drexel.edu

for the unexpected remains one major challenge in the engineering of autonomic systems.

This paper presents an approach to attack that challenge, related to the area of *self-assessment*, a critical pre-requisite for being able to take decisions upon adaptations that an autonomic system may need to enact. Self-assessment requires that the autonomic system must possess means to recognize its operational state (or at least some salient properties of it), and predicate upon that state, in order to evaluate whether it is satisfactory, or should rather be changed.

Many current techniques pursuing self-assessment are – in strict accord with the MAPE-K reference architecture for autonomic systems [10] - analytic in nature: for example, they may monitor the occurrence of certain runtime conditions within the system, as in [2], and try to diagnose their root cause, as in [13].

We propose a non-analytic approach that leverages the concept of application-level utility, an abstraction that captures in a single scalar value the salient properties of an application, with regards to its operational requirements, such as health, value, or quality of service. In practice, our approach for utilitybased self-assessment is three-phased. Firstly, by means of instrumentation, we collect runtime data at two distinct levels: an array of measures sampling the environmental operating conditions for the application; and, in parallel, another set of quantitative characteristics of application-level quality and performance. We then elicit a utility function for the application, through the statistical correlation of the data collected at those levels. Finally, we turn utility into an intrinsic characteristic of the application itself, by using meta-programming techniques to embed the utility function in its runtime libraries; that enables online evaluation of application utility, which in turn lays an objective basis for self-assessment and decision-making.

In the remainder of the paper we describe our method, with the help of various experiments we have carried out, whose results we also present. We discuss the engineering lessons we have learned in the process, with implications in particular to embedding elicited utility functions in legacy applications. Finally, we outline further research opportunities on applicationlevel utility for autonomic computing. But, first, we briefly review the concepts of utility and utility functions, and discuss their usage in Autonomic Computing.

2. Utility in Autonomic Computing

The concept of utility has it roots in Utilitarian philosophy, and its earliest application perhaps in the field of microeconomics, where it indicates the satisfaction generated from owning or consuming a good or service. Computer Science has borrowed the term, and assigned similar semantics to it. For example, in Artificial Intelligence, expected utility has been indicated as a unifying principle for decisionmaking rules [6], thus highlighting how utility expresses the level of satisfaction of one's goals. Utility has been employed in a variety of other computing areas, and even just an overview would easily exceed our space limitations. To offer just a few examples, in communication systems utility can be used to optimize the consumption [3], or pricing [23], of networking resources. In distributed computing, it can for instance arbitrate the allocation of shared data and other resources among concurrent processes [12].

In all cases, it is essential to be able to make the utility abstraction operational. That translates into devising a *utility function* that can capture the definition - and compute values of - utility in the context at hand. A utility function in the form U = f(V) denotes how utility depends on a vector $V[s_0, ..., s_n]$ of attributes that capture the state of the system and its environment; the function **f** maps that - possibly large - vector characterizing the running system to a single scalar value **U**, conventionally in the [0,1] range.

In Autonomic Computing, as mentioned in [11], utility functions become particularly attractive because of their ability to quantitatively attribute points in the system state space. They thus provide an objective and quantitative basis for sound automated decisionmaking, and they can tie those decisions to high-level conceptual requirements, concerns and goals.

A utility function can for example guide automated agents to devise or compare *self-optimization* strategies, in order to ensure certain levels of performance, or even assign monetary value to the operation of a system. A common self-optimization context in which utility has been applied is the automated management of large, multi-application computing infrastructures, like data centers. Walsh et al. [22], and Das et al. [7], among several others, have shown how utility functions can guide resources allocation in a data center, based on the evaluation of the (economic) value of the resources being provisioned for hosted applications, and of those applications for their owners or users. A different selfoptimization scenario is proposed by Cheng et al [5], in which utility evaluation enables the choice among alternative adaptation strategies for changing the configuration of the architecture of a software system at runtime.

Notice that in many current approaches, including the ones above, the definition of the utility functions is not a direct concern for the autonomic system itself. Utility functions are often manually elicited with the help of domain experts or application users [9], or from contractual agreements, such as SLAs [17]. However, objective and (at least partially) automated elicitation is particularly desirable in Autonomic Computing, because it leads to a function that faithfully fits the recorded usage and intended purpose of the target system. Approaches based on negotiation among multiple parties have been used to automate the elicitation of utility functions, limited to the resources allocation scenario [1] [14].

The focus of our work is quite different from what we have described so far, in terms of purpose, as well as scope. We are interested in the non-analytic selfassessment of a single executing application, or even a single recognizable service or task executed by that application. Several other non-analytic approaches exist, based, e.g., on clustering [8] [16], which are able to detect anomalous (i.e. uncommon) behavior. However, those cannot quantitatively attribute anomalies. Application-level utility functions can automatically assign a value to a current state of operation of the application or task, denoting, for instance, its health or performance. In that sense, our work is perhaps most similar to that of Sterritt and Bustard [19], which also establishes a framework for quantitative (although discrete) self-assessment, by means of a multi-level scalar, called a "pulse". However, the definition of pulse levels for an application, with their semantics and thresholds, is left to the implementer.

We are interested, instead, also in the objective elicitation of utility functions, that are tightly linked to observable runtime properties. To that end, we employ a statistics-based approach, which is discussed below.

3. Elicitation of application-level utility

In order to attribute utility semantics to a given application, we conduct a process of correlation and statistical fit between a vector of attributes of the application behavior, and another vector of characteristics of its runtime environment.

To illustrate that process, we briefly discuss hereby the set-up and outcome of a case study on IP teleconferencing, which was the first author's initial foray in this field. In that case study, which is thoroughly described in [18], the collection of application-level data was carried out based upon subjective evaluation by users on their ability and satisfaction in carrying out a variety of communication-related tasks with the support of the VoIP application (user-perceived utility). Those utility values were then correlated to metrics of the network channels used by the application (environment characteristics, namely, bandwidth, latency and packet loss rate), collected from within a testbed environment in which such network conditions could be manipulated and sampled. By throttling those conditions and running a large number of experiments, it was possible to derive mappings between those environment-level characteristics and the utility of the VoIP application as seen by its users. The experiments not only allowed the synthesis of utility functions, but also determined how the sensitivity of each task to the varying conditions is different; hence different utility functions were devised for the same application, on a per-task basis.

Based on the experience of [18], we have started to extend our statistical elicitation technique, to cases in which the utility information is not provided by users. Our goal is to substitute user perception with objective measures of behavioral parameters of the application, as observed, for example, by any client system using that application's services. To achieve that, we instrument the application, and sample a chosen set of salient application-level parameters, in conjunction with environment characteristics. We then try to correlate application-level and environment-level measures: we calculate various moments of the multidimensional graph generated from the environment-level measures, and curve-fit it to the corresponding application-level graph. The equation that produces the best fit becomes our utility function, which allows us to assign a utility scalar to the running application, by simply monitoring the vector of environment- and application-level characteristics included in that equation.

Let us consider, for example, the utility function for downloading a file with an FTP service (FTP utility is discussed in detail as a case study in Section 5.1). For such a task, the most significant application-level measure is throughput, as seen by the FTP client requesting the transfer. That effectively incorporates and abstracts a number of phenomena that occur within the FTP networking environment, such as protocol handshaking and control flow, transfer error, packet loss and retransmission, etc.; but it also depends on other environmental factors, such as the raw throughput and the latency provided by the network socket, and their fluctuations over the time of the transfer. We observe those environment metrics, as well as throughput as seen from the FTP client, for a large number of FTP test runs. We employ a controlled testbed that enables us to collect at corresponding time intervals those tuples (or data points), as well as vary the running conditions from test run to test run, so that we can crawl systematically through the multidimensional operation space constituted by the metrics being observed. We then proceed to synthesize offline the equation for the application-level utility for FTP download, based upon the collected data points.

A critical aspect of our technique is the choice of parameters to be observed, which remains applicationor even task-specific. However, an advantage of our approach over analytic techniques for self-assessment is that - once defined - utility functions do not need to monitor any application-specific events and patterns thereof, nor develop any diagnostic capability to prove/disprove hypotheses that are specific to the application and those patterns. A utility-based selfassessment approach may thus require significantly less domain knowledge than most analytic techniques, which is advantageous, since - as observed for example by Tesauro [20] - the acquisition, modeling and maintenance of domain knowledge in Autonomic Computing is itself an open research issue, and tends to be time-consuming and hard to generalize. Another merit of our approach is that data collection can be fully automated through instrumentation of the application at hand, thus enabling collection in contexts that closely match its natural running conditions ion. Moreover, once our utility function are synthesized, they are already tied to measures that are inherently observable: therefore. the same instrumentation previously used for the elicitation process can be deployed within a utility-aware version of the application, to compute its utility as it operates in the field.

4. Making software applications utilityaware

We have developed a system of code annotations that supports both the data collection and elicitation process, as well as the embedding of the resulting utility function within the runtime code of the application, at the same points where the data was collected. Our annotations intercept the program execution upon entering marked functions in the target application. To the target application, those function calls seem semantically identical, however a background process is triggered by the annotation, which monitors and collects the parameters of interest, and calculates utility on the basis of those parameters.

For our prototype, we selected Python as the target language, for its advanced introspection mechanisms and its ability to foster meta-level programming functionality. We christened our prototype *Aupy*, which stands for "*application utility with Python*".

In a first version of Aupy, our annotations were implemented using a series of Python decorators, associated to method definitions. Those decorators trigger utility-specific code (either for monitoring or utility computation), and specify whether that code should be executed before the target method, after the target method, before and after the target method, or concurrently, that is, in the background while the target method performs its operations. The table in Figure 1 provides a synopsis of Aupy annotations.

| INTENT | SYNTAX | NOTES |
|---|---------------------|---|
| Monitor operation of the annotated code | @monitor(*args) | *args is a dynamic list populated with utility functions. If None is passed, it will use all functions defined with in the same scope as the annotated code. |
| Pre-evaluated utility function | @pre_utility | Evaluation occurs upon entering the annotated code. |
| Post-evaluated utility function | @post_utility | Evaluation happens upon leaving the annotated code. |
| Enveloping utility function | @utility | Evaluation occurs upon entering, and again after leaving, the annotated code. |
| Concurrently evaluated utility function | @concurrent_utility | Evaluation occurs concurrently with the execution of the annotated code. |

Figure 1: Aupy annotations.

In a second version of Aupy, we migrated from Python decorators to Python contexts. Contexts, and the scoping constructs they enable, allowed us to associate utility to specific blocks of code, in addition to entire methods. Annotations at the method level remain very convenient to make legacy systems utilityaware, since it is possible to associate them to - and then seamlessly intercept - also methods defined within and imported from libraries. However, contextbased operations give us better control: as the same primitives for data collection and utility computation are available to the new version of Aupy, we can now work at finer levels of granularity. Another advancement is that utility function can now be composed, thanks to the combined properties of the context construct and Python introspection. That way, we can reuse already defined utility functions to form new ones for a given context. Finally, we can pass dynamic arguments and functions into a context, enabling us to modify the path of execution and thus introduce additional callbacks, embedded contexts, and conditional utility functions. Although similar features could have been engineered also within our decorators, they are much more easily and naturally implemented and integrated in contexts.

Our mechanism for method interception allows for the adaptation of legacy systems to be retrofitted with utility-aware behaviors, but also supports the development of new systems with the same capabilities. This is further illustrated in the case studies that follow.

5. Case studies

5.1 FTP download

The goal of this case study was to validate our approach to elicitation and embedding, as well as the Aupy annotation framework, within a well-understood scenario. We chose the FTP protocol, since it is a wellknown and widely studied system, with a clear principled parameter that can be used to represent the utility of the application, that is, *data throughput*.

This case study was conducted using a standard FTP client written in Python and conforming to the FTP RFC [15]. The FTP client was instrumented to record its application-level throughput, as well as CPU utilization, and RAM usage. From the networking environment, we recorded raw channel throughput and inter-arrival time, as measured on the socket of the client machine. The tasks we profiled with data

collection were the download of one large multigigabyte file (in the following, *FTP-bulk* experiment), and the consecutive download of multiple onemegabyte files (*FTP-small* experiment).

First of all, an annotated version of Python FTP library was constructed. To accomplish that, a subclass of the FTP class in that library was developed, and we used Aupy annotations to mark the subclass methods implementing the FTP commands as code to be monitored. As shown in the example below, the annotated methods simply wrap pass-through statements that invoke the same functionality in the original FTPs library.

| @monitor(None) |
|----------------------------------|
| def ntransfercmd(self, command): |
| return super(SafeFTP, |
| self).ntransfercmd(command) |

Next, a concurrent utility function was developed, to capture and log data at the application as well as at the environment level.

@concurrent_utility def ftp_utility(self): # monitoring code below

The effect of this set of annotations is that, as the monitored transfer command gets executed, Aupy automatically finds all the utility functions defined within the same language scope - in this example only ftp_utility() is available - and run them as prescribed by their @utility annotation.

Finally, we hijacked the original library and substituted it with the annotated version: that is easily accomplished due to the possibility in Python to import library classes under alternate names, to preserve and maintain control of the namespace. The code below overrides the namespace of the original FTP library.

from ftplib import FTP

from aupy_ftp import SafeFTP as FTP

The instrumented FTP client application was then run on a testbed, comprised of two Linux endpoint machines and a BSD machine, which utilized the *dummynet*¹ utility to simulate a variety of network conditions for our test runs. A single test run is a complete file transfer scenario for a dummynet network setup, which is comprised of given values for bandwidth, latency, and packet loss settings. The clocks on all the machines were synchronized before each test run using NTP. A back channel was used for logging and administrative actions, allowing the network channel for the FTP experiments to be fully dedicated to transfers. We carried out a set of 125 test runs and repeated it three times for both the FTP-bulk and FTP-small experiments. During each test run, measurements of environmental and application attributes were recorded each second. We then aggregated the data and performed data interpolation, resulting in an equation that estimates client-side FTP application utility purely on the basis of environment-level metrics, i.e., packet inter-arrival time and channel throughput.

From the process above, we could clearly see how – like in our pervious experience with VoIP [18] - FTP application utility is task-dependent: the equations we synthesized for the bulk-FTP and small-FTP experiments differ significantly, as shown in Figure 2.

| U | $U_{bulk} = a_1 X_1 + a_2 X_2 + a_4 X_4 + a_5 X_5 + b_1 Y_1 + b_2 Y_2 + b_5 Y_5$ | | | | | |
|--|---|----------------|-----------|--|--|--|
| U | $U_{small} = \frac{a}{4} \frac{X}{4} + \frac{a}{5} \frac{X}{5} + \frac{b}{1} \frac{Y}{1} + \frac{b}{2} \frac{Y}{2} + \frac{b}{4} \frac{Y}{4} + \frac{b}{5} \frac{Y}{5}$ | | | | | |
| | Legend | | | | | |
| $X_i : i^{th}$ statistical moment of packet inter-arrival time | | | | | | |
| $Y_i: i^{th}$ statistical moment of socket throughput | | | | | | |
| | FTP-bulk | | FTP-small | | | |
| a_1 | -175150.7079 | a_1 | 0 | | | |
| a_2 | -145163.8736 | a_2 | 0 | | | |
| a ₃ | 0 | a_3 | 0 | | | |
| a_4 | -579.7485 | a_4 | 1028.3839 | | | |
| a ₅ | 2.0136 | a_5 | -157.4976 | | | |
| b_1 | 0.9241 | b_1 | 0.56368 | | | |
| b ₂ | -0.2341 | b_2 | -0.42717 | | | |
| b ₃ | 0 | b ₃ | 0 | | | |
| b ₄ | 0 | b4 | 612.90247 | | | |
| b_5 | -0.17755 | b_5 | 15.18621 | | | |

Figure 2: utility equations for the FTP case study.

Our empirical verification confirmed how estimated utility, i.e., the values of utility computed through the above equations, approximates well the monitored values of throughput for the FTP client (observed utility) on corresponding data points. We show in Figure 3 and Figure 4 the plots obtained from the FTPbulk experiment.

The next step was to embed the elicited functions in the FTP client. For that, we simply had to enhance the function ftp_utility, which previously simply used to log our monitored metrics, with a call to Python code representing the utility equations. We also carried out further validation, since we continued to independently log the FTP client throughput, to keep the observed vs. estimated comparison active online. The same results reported above were confirmed in this phase.

¹ See <u>http://info.iet.unipi.it/~luigi/ip_dummynet/</u>



Figure 3: observed FTP utility.



Figure 4: estimated FTP utility.

5.2 Log explosion

The goal of this case study was to investigate application-level utility to a domain different from protocol-based applications (such as FTP or even VoIP), and to experiment with our technique to solve a real-world problem related to the management of IT infrastructure. The problem at hand regards certain faults that can cause a system to consume a global resource rapidly and unexpectedly. As a consequence, other functioning parts of the system are starved, which usually results in complete system failure. Such a crippling problem can arise for many disparate reasons in a variety of computing systems, in the small as well as in the large.

We have been able to work with one specific form of that problem, occurring within a popular Pythonpowered Web site (more than 1 million unique views per day,): a "log explosion", which caused the log aggregation and rotation subsystem of the Web site infrastructure to be overwhelmed with hard disk writes, rapidly and at unpredictable junctures. System data on those events was collected and analyzed, to discover that a round-robin scheduler in charge of server-side cache replication within a caching cluster would at times fail and remain "stuck" on a given node. Every attempt the scheduler would make to move to the next node would fail, resulting in a log entry. Very quickly the log would exhaust disk space, to the point that the site would no longer be able to accept new connections (which requires logging to be functional). Also attempts to log onto the nodes by other means, like an SSH session, would fail, since there would be no room on the disk to log the session information. Existing monitoring software for the site could not detect this problem in time to avoid a complete system failure. We therefore decided to try to use our framework to recognize log explosions as they happen, without having to investigate or resolve the root cause of that faulty behavior. Our hypothesis was that by monitoring the rates and accelerations of log writes, we could come up with a utility function for the logging facility that would tell us when an explosion was about to occur.

Our testbed for this experiment was simple. One machine was used to run a clean-room re-engineered version of the faulty round-robin scheduler. The system rotated through logical representations of nodes, one of which was injected to cause a fault. Additionally, in the experiment, log information was also piped to standard-out to watch the system live. It was pre-determined that on every twentieth rotation a fault would be triggered on one of the nodes, causing it to become unresponsive to all scheduler requests.

We also embedded Aupy code within the scheduler, to capture the rate of log writes as well as their acceleration rate. Under normal operating conditions. the recorded log writes rate was consistently low, in the order of one write/sec. Our concurrent monitors were set to monitor at the same interval. During repeated experiments, when an explosion occurred, log write rates increased two orders of magnitude in the first second, to roughly 720-780 writes/sec, and again two orders of magnitude in the following second to about 15000 writes/sec. In our particular testing environment, hardware limits for disk write access were reached within the third second. Acceleration appears therefore to be exponential, which shows the extremity of the phenomenon. Consequently, its representation in terms of a utility function produced a step-wise function dropping from 1 to 0 in the event of an explosion.

Whereas we had hoped for a utility profile, which could have possibly provided more preliminary evidence of the build-up of the phenomenon, the stepwise function, once embedded in the system allowed us to recognize log explosions within one second from the triggering of the node fault. We could thus easily put into place a provision to block the round-robin scheduler in time, and stop its runaway logging. We intend to carry out more work on this particular example, to gain further insight on the early assessment of explosive resource consumption problems, .

6. Discussion

6.1 Lessons learned

Our work has provided us with a number of insights on application-level utility for autonomic selfassessment. First of all, we have gained an understanding of a process to elicit utility functions by monitoring the behavior of applications, and their environment. That process is repeatable and requires only a modicum of application-specific knowledge. Although we have experience only with carrying out that process in lab conditions, and in part (i.e. the delicate statistical analysis phase) offline, it seems feasible to consider it as a basis for working toward online, automated synthesis of utility functions. We expand on this matter as a possible direction for future research in Section 6.2.

Our case studies have shown how utility at the application level is task-dependent, and a number of consequences descend from that. Since an array of utility functions apply to the same system, and, in some cases, such as our case study on FTP download, to the same application feature, having elicited those distinct utility functions is not *per se* sufficient: one must also have the means to select them opportunistically, based on the context of the task being performed. That has engineering implications, and has become one of the requirements driving our Aupy framework. Our latest prototype allows to associate multiple (possibly concurrent) utility functions to the same code block. It also accounts for conditional triggering and utility function composition, via the construct of Python contexts.

The task-dependent nature of application utility also helps understanding how our method fits naturally well certain specific classes of applications. One example is constituted by communication-oriented systems, which are typically based upon protocols that codify and restrict what the system is intended for. That facilitates the extraction the system's salient features, and can guide the elicitation of appropriate utility functions for the application tasks that reify those features. Similar considerations often apply to infrastructure-level facilities that - like in on our case study on server logging – are characterized by a limited set of specific, self-contained services they provide to the computing environment as a whole. In the case of other classes of applications, e.g., enterprise systems, the synthesis of utility functions across the spectrum of all user tasks enabled by their possibly very diverse, heterogeneous services can become unfeasible, in particular in the absence of automated support in the elicitation process.

An advantage of the fine-grained utility abstractions required and at the same time enabled by our framework is a viable approach to comparable, interoperable and composable definitions of "health" for different autonomic applications, which descends from the unit-less and summative properties of utility functions. It is well known how this applies at a macroscopic level, e.g., for multiple applications operating within the same provisioning infrastructure (leading to a view of utility for the whole computing environment, as discussed in [22]). It applies as well at the micro-level of different tasks being performed within the same application: for example, related to our FTP case study, the utility of the FTP download service as a whole could be expressed as a combination of the utility values recorded for bulk downloads and small download tasks as they occur. Things are analogous in the increasingly common case of component-based applications, or even systems-ofsystems, in which different pieces of legacy, COTS or otherwise third-party functionality are integrated into a new application. If all components could be characterized with embedded utility functions, the utility of the overall composite system could be derived by combining the utility values recorded for each of the various components.

Lastly, we have been able to test the limits of our hypothesis that the utility abstraction can lead to nonanalytic self-assessment. Domain-specific knowledge plays a role in our approach, but only in the initial data collection phase, for the selection of the basic characteristics of the application or task that must be monitored. Past that phase, the whole process of function elicitation and embedding, as well as utility computation, remains application-agnostic. We see this as a major factor enabling the assessment of the application under scrutiny, no matter how unexpected or unpredictable its state of operation can become.

6.2 Ongoing and future work

A central part of our approach is the process for the elicitation of utility functions, which requires the collection of a large amount of data and its statistical analysis. Although data collection is automated, this is currently a time-consuming process that occurs in the lab, by means of a testbed environment that needs to be configured for each application. Moreover, the statistical analysis occurs offline, and can be rather labor-intensive. This is an issue that must be resolved, in order to use utility functions for self-assessment extensively (that is, for a diverse range of applications), and at the correct level of granularity (that is, for the distinct features and tasks enabled by a given application). We are now considering how to attack that issue.

First of all, we want to move from in-the-lab to inthe-field data collection. With Aupy, it is easy to deploy in the field an instrumented version of the application under scrutiny, so that its various features can be exercised in regular working conditions. However, in that case, it may be difficult to produce an exhaustive map of evenly distributed data points, to be fed to our statistical correlation analysis. It is likely that many of the collected data points would tend to gather in one or more dense areas within that space, with other data points constituting outliers or operational anomalies. Consequently, it might be unfeasible to synthesize an equation that is valid outside of the main operation areas, and which covers and accurately attributes utility to anomalous data points. On the other hand, we could identify and tell apart different operation areas by analyzing in-the-field data with clustering techniques – as shown for instance by Quiroz et al. [16] - and possibly characterize those operation areas in terms of different application features being exercised. That would easily lead to the elicitation of distinct utility functions for each of them. Drawing an example from our FTP case study, tasks like downloading of a single large multi-GB file, and downloading of multiple 1 MB files would correspond

to two distinct, recognizable data point regions, and could be treated independently from one another.

We also plan to leverage some AI-based techniques. to advance towards automation of utility function elicitation. For example, cooperative negotiation has been already applied to Autonomic Computing contexts, in which overall utility is not known or cannot be expressed a priori, and must be elicited incrementally from multiple interested parties [1] [14]. To apply and adapt those earlier results to our context, we could employ multiple evaluating agents, each defining its utility in terms of a single axis of the multidimensional space being monitored. Each agent would compute its local utility values for a number of sample requests, and the aggregation of the various utility perspectives thus generated would be used to compute the overall utility function. Further insight on this matter could come from the field of automated learning. We plan to experiment with supervised and unsupervised learning techniques, to help deriving utility functions from the potentially large and highdimensional space monitored through Aupy.

Finally, since a utility function can be regarded as a multi-dimensional map of application value, in which the axes represent environmental as well as application attributes, and assuming a management infrastructure that is able to actuate some subset of those attributes (at either level), it is conceivable to elaborate actionable trajectories within that map, to move an application from a low-utility state towards a higherutility state. A somewhat similar idea has been pursued in [4]. To efficiently calculate those trajectories, we could again leverage machine learning techniques, in particular Reinforcement Learning (RL). RL has been successfully used for Autonomic Computing in the recent past to synthesize policies for decision-making. aiming once again at self-optimization [20] [21]. A key concept in RL is "reward", a scalar that valuates the observed consequence of a decision, and which the learning agent making decisions aims to maximize. The concept of utility naturally maps onto RL rewards; thus, having elicited a utility function, it is easy to compute rewards for trajectories within the state space of the application, and segments within a trajectory.

With that development, utility-based methods could be applied in Autonomic Computing not only to nonanalytic self-assessment (i.e., within the reactive/introspective "Monitor+Analyze" portion of the MAPE control loop), but also to the equally important area of non-analytic, automated decisionmaking. Such a technique could thus enable the design of provisions for the "Plan+Execute" half of the MAPE loop devoted to proactive adaptation, which could cope with unexpected events and potentially conditions.

7. Conclusions

This paper has presents a method and a tool for the elicitation of application-level utility functions, which can be subsequently employed for autonomic selfassessment. Our work has provided us with an understanding of an elicitation process based on the statistical analysis of measurable data that is collected at two levels: the application under scrutiny, and the running environment hosting that application. We have also acquired know-how on the engineering of a framework for data monitoring and collection, as well as the embedding of elicited utility functions in the runtime code, which leads to seamless computation of utility as the applications operates. We have developed the Aupy prototype for such a framework, and experimented with it in two case studies from diverse application domains. Through those case studies we have acquired insight on the task-based nature of application-level utility, which has prompted us to include fine-grained, block-wise support for utility functions in our framework.

This work shows how application-level utility is a viable abstraction for the non-analytic self-assessment of autonomic systems, and can be made operational for newly developed as well as legacy systems.

Finally, this can be a preliminary step to conduct further research on application-level utility in Autonomic Computing, regarding the fully automated, in-the-field elicitation of utility functions, as well as the possibility to leverage utility information for automated and non-analytic decision-making on application adaptation.

8. Acknowledgements

The authors would like to thank the other members of the Software Engineering Research Group (SERG) at Drexel University. In particular, we are grateful to Prof. Spiros Mancoridis, and to students Max Shevertalov and Ed Stehle, who have been instrumental in initiating and carrying out the work leading to the reported results.

References

- [1] C. Boutilier, et al., "Cooperative Negotiation in Autonomic Systems using Incremental Utility Elicitation," Proc. 19th Conference on Uncertainty in Artificial Intelligence 2003, pp. 89-97.
- [2] M.Brodie et al. "Quickly Finding Known Software Problems via Automated Symptom Matching", in

Proceedings of the 2nd International Conference on Autonomic Computing (ICAC 2005), June 2005.

- [3] Z. Cao and E.W. Zegura, "Utility max-min: an application-oriented bandwidth allocation scheme," in Proceedings of INFOCOM'99, IEEE, 1999, pp. 793-801.
- [4] M. Karlsson, and M. Covell, "Dynamic Black-Box Performance Model Estimation for Self-Tuning Regulators", in Proceedings of the 2nd International Conference on Autonomic Computing (ICAC 2005), June 2005.
- [5] S.W. Cheng, et al., "Architecture-based self-adaptation in the presence of multiple objectives," in Proceedings of the ICSE International Workshop on Self-adaptation and self-managing systems (SEAMS 2006), ACM Press, 2006.
- [6] F.C. Chu, and J.Y Halpern, "Great Expectations. Part II: generalized expected utility as a universal decision rule", *Artificial Intelligence*, Elsevier, November 2004, 159(1-2):pp. 207-229.
- [7] R. Das, I. Whalley, and J.O. Kephart, "Utility-based collaboration among autonomous agents for resource allocation in data centers", in Proceedings of the 5th Intl. Joint Conference on Autonomous Agents and Multiagent Systems, Hakodate, Japan, 2006.
- [8] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles", in 23rd International Conference on Software Engineering (ICSE), Toronto, Ontario, Canada, May 2001
- [9] D.E. Irwin, et al., "Balancing Risk and Reward in a Market-Based Task Service," in Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13 '04), IEEE, 2004, pp. 160-169
- [10] J.O. Kephart, and D.M. Chess, "the Vision of Autonomic Computing", IEEE Computer 36(1): 41-50, January 2003
- [11] J.O. Kephart, and R. Das "Achieving Self-Management via Utility Function", IEEE Internet Computing 11(1):40-48, January 2007.
- [12] J.F. Kurose, and R. Simha, "A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems", *IEEE Transactions on Computers*, May 1989, 38(5):pp705-717.

- [13] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller, "Problem diagnosis in large-scale computing environments", in Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL., USA, 2006.
- [14] R. Patrascu, et al., "New Approaches to Optimization and Utility Elicitation in Autonomic Computing," Proc. National Conference on Artificial Intelligence, 2005, pp. 240-245.
- [15] J. Postel, and J. Reynolds, "File Transfer Protocol (FTP)", IETF RFC 959, October 1985, available at: http://www.ietf.org/rfc/rfc959.txt
- [16] A. Quiroz, et al., "Robust Clustering Analysis for the Management of Self-Monitoring Distributed Systems," Journal of Cluster Computing, November 2008
- [17] M. Salle and C. Bartolini, "Management by contract," Proc. Network Operations and Management Symposium (NOMS 2004), IEEE/IFIP, 2004, pp. 787-800.
- [18] E. Stehle, M. Shevertalov, P. deGrandis, S. Mancoridis, and M. Kam, "Task Dependency of User Perceived Utility in Autonomic VoIP Systems", in Proceedings of the International Conference on Autonomic and Autonomous Systems, (ICAS 2008), Gosier, Guadeloupe, March 2008
- [19] R. Sterrit, and D. Bustard. "A Health-Check Model for Autonomic Systems Based on a Pulse Monitor", Knowledge Engineering Review, 21(3):195-204, September 2006.
- [20] G. Tesauro, "Reinforcement Learning in Autonomic Computing. A Manifesto and Case Studies," IEEE Internet Computing, 11(1):22-30, January 2007..
- [21] G. Tesauro, et al., "On the use of hybrid reinforcement learning for autonomic resource allocation," Journal of Cluster Computing, vol. 2007, no. 10, 2007, pp. 287-299.
- [22] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das, "Utility Functions in Autonomic Systems", in Proceedings of the International Conference on Autonomic Computing (ICAC 2004), New York, NY, USA, May 2004.
- [23] X. Wang and H. Schulzrinne, "Pricing network resources for adaptive applications in a differentiated services network," in Proceedings of INFOCOM 2001, IEEE, 2001, pp. 943-952.