Auto-Adjusting Self-Adaptive Software Systems

Published in the Proceedings of the 15th IEEE International Conference on Autonomic Computing (ICAC), pp. 181-186, 2018

Zoltán Ádám Mann and Andreas Metzger
paluno – The Ruhr Institute for Software Technology, University of Duisburg-Essen, Essen, Germany

***Abstract*—Self-adaptive systems can cope with changes in their operating environment by modifying their structure and behavior at run time. Different kinds of changes pose different requirements on how the software should adapt: some changes may require an immediate adaptation, whereas others do not, leaving more time to find the most suitable action. To address different kinds of changes, we introduce auto-adjustment, which works by quickly assessing changes in terms of the resulting requirements on the adaptation logic (e.g., their criticality or urgency), and adjusting the adaptation logic accordingly. Thereby, auto-adjustment allows dynamically considering the trade-off between adaptation speed and adaptation quality. Experiments with an autonomic cloud resource allocation system show that auto-adjustment leads to an improved trade-off between conflicting system goals: by allowing 0.3% higher energy consumption, the number of server overloads can be reduced by 68%.**

***Index Terms*—self-adaptive systems, on-line reconfiguration of the adaptation logic, cloud resource management**

I. Introduction

Today's software systems must operate in highly complex and dynamic environments [1], [2]. Self-adaption has been proposed to cope with such dynamic environments [3]. Self-adaptive software continually monitors its environment to detect changes. If needed, the software modifies its own structure and behavior at run time to continue meeting its functional and quality objectives. Self-adaptive software exhibits improved resilience to failures, better resource usage, and higher performance [4].

However, problems arise if adaptations are done in a way that is not fit for purpose in a given situation. For example, some adaptation actions may take more time than others [5]. Not taking this into account may lead to poor adaptation decisions; e.g., choosing a lengthy adaptation action when a critical situation calls for immediate action. Similar problems may arise if the procedure to determine the best adaptation action takes too long; e.g., many reasoning techniques proposed for computing adaptation actions are computationally expensive [6]. If the environment is not monitored with appropriate frequency, this can also lead to problems; e.g., if sensor readings arrive every t seconds but significant changes can happen in less than t seconds, the adaptation logic may not be able to timely observe and handle these changes [7].

The required frequency for executing the adaptation logic and the allowed duration for computing adaptation actions depend on the specific system and context, and may even change dynamically. For instance, a sudden massive increase in the load on a web application requires a quick reaction to avoid a serious overload of the system, whereas an adaptation of the same system is less urgent if the load decreases [8].

As an example, Fig. 1 shows a real workload of a cloud service. In different periods of time, the workload exhibits different dynamics: sudden decreases, slow increases etc. These periods impose different requirements on the self-adaptation of the system. Thus, in such highly dynamic environments, it is not sufficient to only adapt the system: the adaptation logic itself must be dynamically modified to address the changing adaptation requirements. Modification of the adaptation logic should be fast to enable timely reaction to changed dynamics (e.g., a change from decreasing to increasing load).

Existing approaches to self-adaptation do not solve this problem. Running the adaptation logic with the *highest possible speed and frequency* prohibits computationally expensive optimizations, even if such optimizations would be sometimes possible. *Hierarchical approaches*, such as the three-layer architecture [9], change the adaptation logic only after observing that the adaptation logic is not appropriate in the current context – which may be too late. *Online learning* improves the adaptation logic by a trial-and-error process, which may require many observations to converge [10], with negative consequences on the live system [11].

To address the different kinds of environment changes that a self-adaptive system may face, we introduce the notion of auto-adjustment. Auto-adjustment augments the adaptation logic of a self-adaptive system. It can directly trigger quick reactions if needed, which sets it apart from most of the related approaches. To be fast, auto-adjustment is based on a quick and coarse-grained assessment of the changes currently faced

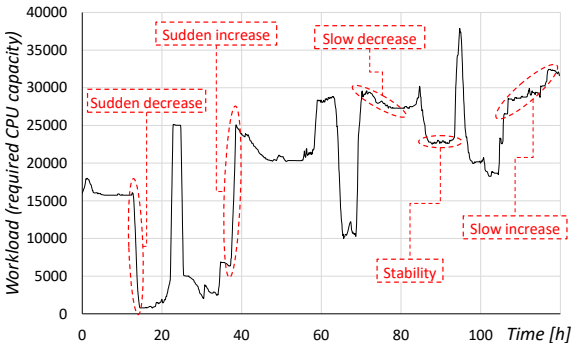

Fig. 1: Example workload

by the system. As a result of the assessment, the parameters of the self-adaptive system's adaptation logic are set to values that are appropriate for the currently faced changes.

Auto-adjustment can be used to differentiate between modes of operation for the adaptation logic. As a simple example, the adaptation logic may work in *normal mode*, in which the best possible adaptation is aimed for, even if that takes some time. Besides, there can be an *emergency mode*, in which a good enough adaptation must be found quickly to cope with time-critical situations. Auto-adjustment makes a situational choice among the available modes of adaptation. Thereby, auto-adjustment allows dynamically considering the trade-off between adaptation speed and adaptation quality.

II. Running example

We consider a data center with m servers. Each server has given capacity along multiple resource dimensions, e.g., CPU and memory. A server that is switched on may host virtual machines (VMs). At a given point in time, each VM has specific resource needs (e.g., in terms of CPU and memory). The load of a server for a specific resource type is the sum of the needed resources of the VMs that it hosts. If the load of a server exceeds its capacity for at least one resource type, then the server is overloaded. Overloading a server results in performance degradation of the hosted VMs and in turn to violations of service level objectives; hence, server overloads should be avoided as much as possible [12]. One of the main cost drivers in data center operation is the power consumption of running servers. To minimize power consumption, the number of servers that are on should be minimized.

The load on cloud services varies over time, leading to oscillations in the VMs' resource needs (cf. Fig. 1). To react to changes in the workload, VMs can be migrated between servers. When the load of some VMs is low, they can be consolidated to fewer servers. Unused servers can be switched off, thus saving energy. When the load of the VMs is high, more servers need to be switched on so that the VMs can be spread across more servers, thereby avoiding server overloads.

To sum up, data center operators must continually re-optimize the mapping of VMs to servers to react to changes in the workload. The considered adaptation actions are switching servers on or off and migrating VMs between servers. The objectives are: (i) minimizing power consumption, (ii) avoiding server overloads. The resulting optimization problem is computationally challenging to solve optimally, but different algorithms exist to solve it heuristically [13], [14].

III. Problem Definition

In self-adaptive systems, the quality of the adaptation process has several important characteristics. E.g., how quickly does the system adapt to changes? What is the maximum frequency and maximum magnitude of change that the system can cope with? How stable is the self-adaptive system against short temporary changes? In practice, the adaptation logic of a self-adaptive system is designed based on assumptions about the required quality of the adaptation process for a given application. Often, these assumptions are implicit and are based on experience and common sense [4], [8]. In the running example, if the cloud provider decides to re-optimize the placement of VMs every t seconds, this is based on the assumption that on a time scale smaller than t, no big change in the workload takes place that would need faster reaction. A design decision that in one iteration of the adaptation logic at most one server is turned on, is based on the assumption that the changes in the workload between two consecutive invocations of the adaptation logic can be absorbed by the addition of a single new server.

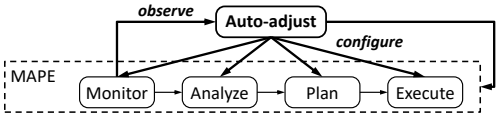

Fig. 2: Augmenting the MAPE loop with auto-adjustment

If changes are more frequent and/or have bigger amplitude than assumed, the adaptation logic may not react adequately to the changes. In the running example, multiple problems can arise if the changes in the workload are not in line with the assumptions underlying the operation of the adaptation logic. For instance, if the adaptation logic is not executed with sufficient frequency, the adaptation logic may detect only with delay that the workload is rising. Or, if the workload is rising more rapidly than assumed in the design of the adaptation logic, the adaptation logic may run analysis and planning techniques that take too long for a timely reaction.

If the adaptation logic is not working in line with the current context dynamics, this may be harmful. It is not easy to design the adaptation logic in such a way that it can effectively cope with different kinds of dynamics. As shown in Fig. 1, different types of context changes may exist, posing very different requirements on the adaptation logic.

The problem that we address can be summarized as follows: the dynamics (frequency and magnitude of changes) of the context of a self-adaptive system may change over time, leading to different requirements towards the adaptation logic. The operation of the adaptation logic should be in line with the current context dynamics. This requires a systematic way to detect the current context dynamics and reason on how the adaptation logic should operate to cope with it.

IV. The auto-adjustment approach

To tackle the issues described in Sec. III, we introduce an approach called auto-adjustment. We use the well-known MAPE reference model as a basis [3]. MAPE separates adaptation into four steps: Monitoring, Analysis, Planning, and Execution. As shown in Fig. 2, our approach adds a further step called *Auto-adjust*. The Auto-adjust step continually assesses the situation using the data provided by the Monitoring step and automatically configures the MAPE steps so that their behavior is always aligned with the current situation. For the latter, the adaptation logic should be configurable via appropriate parameters (like in [15]). To react quickly to

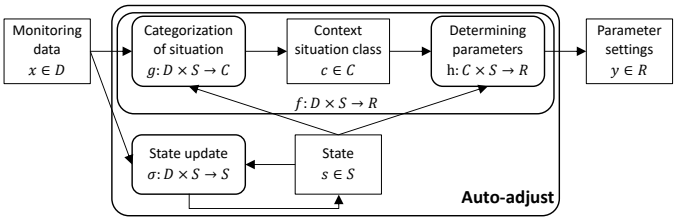

Fig. 3: Proposed structure of the Auto-adjust step

TABLE I: Context situation classes – example

Class	Characterization	A	P	F
C_1	Load is quickly growing	A_1	P_1	F_2
C_2	Load is decreasing; there are overloaded servers	A_2	P_2	F_1
C_3	Load is decreasing; there are no overloaded servers	A_3	P_3	F_1
C_0	None of the above applies	A_4	P_4	F_1

changes in the dynamics of the environment, auto-adjustment must be fast. Therefore, it uses a quick and coarse-grained assessment of the current state. Then, it just adjusts parameters of the adaptation logic.

1) Interplay with the MAPE steps: As shown in Fig. 2, the Auto-adjust step gets information from Monitoring and may set parameters of each MAPE step or the MAPE loop as a whole. Fig. 2 shows logical connections between the MAPE steps and the Auto-adjust step; it does not specify the order of execution of the steps. There are multiple possible scenarios for the order of execution of the steps: For example, Auto-adjust may wait until the monitor step is finished to get the results of monitoring, but it is also possible that Auto-adjust needs only a part of those results which can be made available earlier. Likewise, Analyze may start after Auto-adjust finished, or they can run in parallel and Auto-adjust may decide to abort Analyze if it turns out that an urgent reaction is needed.

2) Inputs: Auto-adjust gets a subset of the output of Monitoring that also Analyze is based on. This way, no further sensors are needed for Auto-adjust; the existing sensors that are anyway used by the adaptation logic can be reused. Whether the whole Monitoring data for Analyze is also passed to Auto-adjust or only a subset thereof, depends on the time requirements. If the Monitor step involves lengthy processing of sensor data, then it may be necessary to feed Auto-adjust with raw or partially processed data instead so that it does not have to wait for that processing to finish. Otherwise, all the Monitoring output can be provided to Auto-adjust.

3) Outputs: The outputs of Auto-adjust control the way the adaptation logic works. For this purpose, the operation of the adaptation logic should be controllable through a set of parameters. We assume that reconfiguring the adaptation logic by changing parameters is fast, e.g., by switching between available algorithms. Lengthy reconfigurations like synthesizing a new controller or re-compiling a program should be avoided to be able to react quickly to critical changes.

Each MAPE step can have parameters. E.g., the number of past observations to take into account may be a parameter for Analyze. Further parameters may relate to the MAPE loop as a whole, e.g., the frequency of carrying out the MAPE loop.

4) Processing: Let the input of Auto-adjust be a vector x of monitoring data from some domain D (e.g., if monitoring consists of n real-valued monitoring signals, then $D = \mathbb{R}^n$). The output of Auto-adjust is a vector y of parameter values in a range R (e.g., if there are k binary parameters, then $R = \{0,1\}^k$). Moreover, Auto-adjust may have a state $s \in S$, based on previous inputs. Then, the operation of Auto-adjust can be described as a function $f : D \times S \to R$, coupled with an internal state updating function $\sigma : D \times S \to S$. To compute f, we propose decomposing it into two sub-steps and thus sub-functions g and h with $f = h \circ g$ (see Fig. 3):

- g: Coarse-grained categorization of the current situation and the type of change that is going on, based on the monitoring input. Let C be a set of *context situation classes*; then $g : D \times S \to C$ maps each possible monitoring input to a context situation class.
- h: Determining appropriate parameter settings based on the identified situation. This can be formalized as a function $h : C \times S \to R$, mapping each context situation class to the appropriate parameter settings.

To be fast, the first sub-step may be carried out using thresholds on the observed values, and the second sub-step using event-condition-action rules. The thresholds and the rules may be static, but they may also be changed dynamically (e.g., by learning), as long as this does not affect the speed of Auto-adjust (e.g., by performing the learning off-line).

V. Application to the running example

1) Context situation classes: Deriving the relevant classes of context situations (C) should be based on the goals of the adaptation logic, the opportunities that could be used to achieve those goals, and the situations that should be avoided. In our case, VMs should be consolidated to as few servers as possible. Consolidation opportunities arise mainly if the load of VMs is decreasing. Second, server overloads should be avoided or eliminated quickly. Server overloads arise if the load of VMs increases or because of the overhead of migrations. Third, to keep the number of VM migrations low, VMs with non-correlating load patterns should be consolidated to the same server because they can co-reside on the same server for a long time without causing overloads [16].

On this basis, we can derive the context situation classes, $C = \{C_0, \ldots, C_3\}$ (see Table I, first two columns).

2) Needed monitoring input: The context situation classes in Table I can be identified based on two types of information: growth rate of the load and existence of overloaded servers. To compute function g, we thus need to monitor this information. Monitoring has to collect information for the adaptation logic anyway on the load of VMs and servers. From that, it can be determined whether there are overloaded servers. The load of different VMs may experience different growth rates; we use an aggregate metric that shows whether there is an overall

growth in load: the ratio of VMs whose load grew by at least a given percentage since the last measurement.

3) Required behavior: In C_1, the risk of server overloads is high. The adaptation logic must react quickly and prevent or mitigate overloads. The analyze step should be restricted to identifying the servers most prone to overload. Planning should be restricted to quickly finding a (not necessarily optimal) plan for relieving the highly loaded servers with migrations. The MAPE loop should be restarted soon to check if the problem is solved or further actions are necessary.

In C_2, the adaptation logic should quickly relieve overloaded servers. Analysis should be restricted to identifying the affected servers, while planning should be restricted to quickly devising a plan to relieve those servers with migrations.

In C_3, no immediate action is required. The adaptation logic should aim for long-term goals. Analysis should assess which subsets of VMs have load patterns that are compatible for long-term co-residence. Planning should find the best possible[1] allocation of VMs to servers based on the analysis results.

In C_0, only small changes occur between subsequent MAPE cycles, hence the adaptation logic should address those changes. Analysis should identify the changes, while planning should address them with appropriate local actions.

4) Parameters: The Analyze and Plan steps need to be configured, as well as the frequency of the MAPE loop. For Analyze, a parameter A is needed to choose among four different behaviors:

- A_1: Identify servers that are either overloaded or likely to become overloaded soon
- A_2: Identify overloaded servers
- A_3: Assess which subsets of VMs have load patterns that are compatible for long-term co-residence
- A_4: Identify significant changes since the last invocation

For the Plan step, a parameter P is needed to choose among four different behaviors:

- P_1: Quickly find a plan for relieving highly loaded servers
- P_2: Quickly find a plan to relieve overloaded servers
- P_3: Find the globally best allocation of VMs to servers
- P_4: Devise a set of local actions to address changes

The parameter F, for defining the frequency of the MAPE loop, has two values: F_1 for normal execution, and F_2 for more frequent execution of the loop. Table I shows the mapping between the context situation classes and the parameter settings (i.e., the function h).

VI. Experimental Evaluation

1) Implementation: We based our implementation on CloudSim, a widely-used cloud simulator [18]. In CloudSim, adaptive resource management is realized by a VM consolidation algorithm which combines analysis and planning. It determines a set of migrations to (i) relieve overloaded servers and (ii) consolidate VMs to as few servers as possible. CloudSim contains a built-in VM consolidation algorithm, which is a fast heuristic [19]. We created an alternative VM consolidation algorithm using integer linear programming (ILP) [20]. The ILP-based algorithm is significantly slower than the built-in heuristic, but usually gives better results. When the ILP-based algorithm does not find a valid solution within the available time budget, the built-in heuristic is used as fallback.

Based on these two VM consolidation algorithms, we implemented a simplified variant of the scheme described in Sec. V. The simplification relates to the fact that the available VM consolidation algorithms are not decomposed according to the MAPE model, so that fine-grained control of the MAPE steps is not possible. We use two context situation classes: (i) emergency mode, in which the adaptation logic must quickly react to prevent server overloads and (ii) normal mode, in which there is sufficient time to aim for a placement of the VMs that is as good as possible. For adjusting the adaptation logic, two parameters are provided: one for the choice between the two VM consolidation algorithms and another to choose the re-optimization interval, i.e., the time before the adaptation logic is invoked again.

2) Experiments: We simulated a cluster of 100 servers, hosting 500 VMs. The servers belong to three types, with CPU capacities of 2000, 4000, and 8000 MIPS (million instructions per seconds). The VMs' requested CPU size ranges from 200 to 1500 MIPS, and their actual CPU size is always defined as percentage of their requested size, as explained below. Re-optimization is normally carried out every 5 minutes (i.e., $T_{\text{normal}} = 300s$); for emergency mode, it is set to $T_{\text{fast}} = 150s$. The ILP-based algorithm is given a time budget of 60 seconds; the execution time of the heuristic algorithm is negligible (below 1 second). We performed two experiments, both lasting 5000 seconds. In each experiment, three approaches were compared: (i) always using the heuristic algorithm, (ii) always using the ILP-based algorithm, (iii) the auto-adjusting approach.

In the first experiment, the load of the VMs decreased from 90% to 10% in the first 2000 seconds, followed by constant load for 1000 seconds, and then increased back to 90%. As Fig. 4a shows, the number of active servers closely follows this trend for all three approaches, showing that all algorithms react to the changes. When the workload decreases, the ILP-based and the auto-adjusting approaches yield very similar results; the heuristic leads to slightly more active servers. When the load is constant, all three approaches lead to practically the same results. When the workload increases, the heuristic and the auto-adjusting approaches increase the number of active servers faster, so the ILP-based approach leads to the least active servers. Fig. 4b shows the cumulated number of server overloads, which is 0 when the load decreases or remains constant. When the workload starts to increase, each approach leads to some overloads. The auto-adjusting approach is the fastest to get into a mode in which it does not generate further server overloads. Thus, the other two approaches lead to significantly more server overloads. The ILP-based approach leads to the highest number of server overloads.

[1]Since finding the optimal allocation is computationally intractable [17], it is still necessary to use a heuristic or a timeout, but in this case, planning is allowed to take significantly longer than in the two preceding cases.

(a) Development of the number of active servers over time

(b) Cumulated number of server overloads over time

(c) Situation determined by the auto-adjustment approach

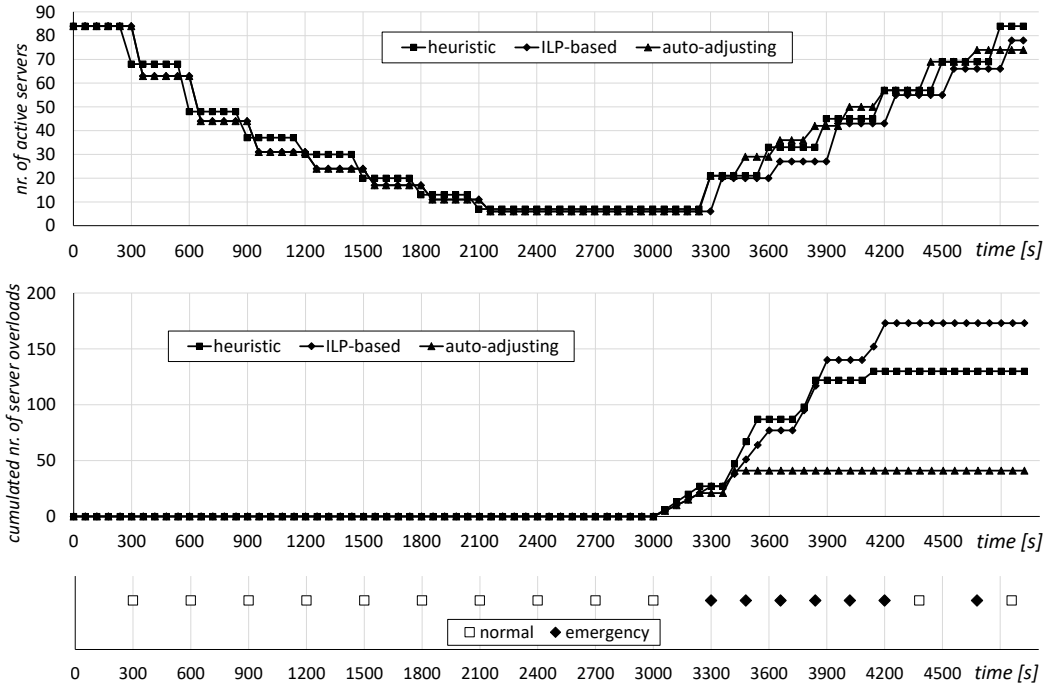

Fig. 4: Results of the first experiment

Fig. 4c shows how the auto-adjusting approach classifies the situation during its invocations: normal mode is active when the workload decreases or remains constant; when the load starts to increase, emergency mode is activated, leading to more frequent invocations of the MAPE loop.

In the second experiment, the workload has similar shape, but the changes are much smaller: the load starts again at 90%, but decreases only to 80%, remains constant, then increases again to 90%. The number of active servers again closely follows the workload for all three approaches. The number of active servers is similar for the ILP-based and the auto-adjusting approaches, while it is consistently higher for the heuristic. The number of server overloads is 0 for all approaches. The situation determined by the auto-adjusting approach is normal mode throughout the second experiment.

3) Analysis of the results: Auto-adjustment activated emergency mode when the load was quickly rising (right half of Fig. 4), while it remained in normal mode in all other cases. This was a sensible choice: when the load increased quickly, emergency mode allowed quick and frequent re-optimization of the VM placement, thus leading to significantly less server overloads. In all other cases, this was not necessary (there were no server overloads), and using normal mode made it possible to fully utilize the consolidation possibilities, thereby using less active servers.

The results of the three approaches, aggregated over the two experiments, are summarized in Table II. The ILP-based approach leads to the best energy consumption, closely followed by our approach (the difference is only 0.3%); the heuristic leads to more than 16% higher energy consumption.

TABLE II: Aggregated results

Approach	Consumed energy [kWh]	Nr. of server overloads
heuristic	62.64	130
ILP-based	53.87	173
auto-adjust	54.03	41

Concerning server overloads, our proposed approach achieves the best results, leading to 68% and 76% less server overloads than the heuristic respectively the ILP-based approach. This shows that auto-adjustment can lead to substantial benefits.

VII. Related work

The problem that a fixed adaptation logic may not be appropriate in all situations has been recognized [21]. To address this problem, several authors suggested adding extra layers to a self-adaptive system's adaptation logic. These additional layers monitor the adaptations and modify the adaptation logic if adaptation performance is unsatisfactory. The first such approach was the *three-layer architecture*, in which the managed element (the lowest layer) is adapted using reactive plans by the adaptation logic of the middle layer, while the task of the upper layer is to devise new plans [9]. The three-layer model was later refined by other authors [22].

Other hierarchical approaches use *online learning* to improve the performance of the adaptation logic; e.g., by learning models that capture knowledge about the environment [23]. A concern with this approach is that while online learning is converging, the self-adaptive system may execute ineffective

adaptations, which can have negative consequences, because the ineffective adaptations happen in the live system [11].

These hierarchical approaches are based on two critical assumptions. First, it is assumed that the environment is largely stable – in the sense that changes are small – for longer periods of time, with occasional transitions between stable states. This is true in many cases; however, in highly dynamic settings without lengthy epochs of stability, these approaches are not appropriate. In the three-layer architecture, the time it takes to recognize that the lower layers cannot deal appropriately with the current situation and to wait for the upper layer to elaborate new adaptation plans may lead to non-acceptable delays. In online learning, the time it takes to converge to an acceptable solution may not be acceptable.

The second assumption is that, during the transient periods after changes in the dynamics of the environment, suboptimal adaptations can be tolerated. This implicit assumption is mirrored by the fact that in the evaluation of previous approaches, either only aggregated metrics were used to judge the performance of the methods, or transient periods were simply excluded. Again, this assumption may not hold in some domains, first because of the already mentioned dynamism (practically, there may only be transient periods), and second because in some systems, a bad adaptation can have wide-ranging negative consequences.

In contrast, our auto-adjusting approach recognizes in time the need for changing the adaptation logic and quickly reconfigures it appropriately. Therefore, our approach is better suited for highly dynamic environments.

Several authors proposed adopting methods from control theory to engineer self-adaptive software because of the similarities between the two fields and the benefits of the mathematically well-founded approaches of control theory [24]. However, guaranteeing desirable properties like stability or quick responses by means of control theory requires strong assumptions: e.g., that there is enough time for converging to a stable state, the frequency of changes between stable states is bounded, and disturbances and transients are secondary and can be identified [7]. As discussed above, such assumptions may not hold in highly dynamic settings.

Particularly relevant classes of systems in control theory are *auto-tuning systems* and *adaptive control*, which are capable of optimizing their own parameters to optimize the fulfillment of an objective function [25], [26] However, auto-tuning for self-adaptive software systems is challenging and has not yet been successfully achieved [7].

VIII. Conclusions

In this paper, we proposed auto-adjustment as a means to keep the adaptation logic of a self-adaptive system in line with the changing context dynamics. With auto-adjustment, it is possible to switch on the fly between available modes of adaptation, satisfying different requirements on the adaptation logic. A case study of an adaptive cloud resource management system showed that dynamically choosing from just two different modes of adaptation can lead to an improved trade-off between system goals.

Interesting paths for future research include the combination of auto-adjustment with machine learning and prediction techniques to further improve its potential.

Acknowledgments. This work received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements no. 731678 (RestAssured) and 780351 (ENACT).

References

[1] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Commun. ACM*, vol. 55, no. 9, pp. 69–77, 2012.

[2] Z. A. Mann and A. Metzger, "Optimized cloud deployment of multi-tenant software considering data protection concerns," in *Proc. CCGrid*, 2017, pp. 609–618.

[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[4] J. Cámara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. Schmerl, and R. Ventura, "Incorporating architecture-based self-adaptation into an adaptive industrial software system," *J. Syst. Softw.*, vol. 122, pp. 507–523, 2016.

[5] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl, "Proactive self-adaptation under uncertainty: A probabilistic model checking approach," in *Proc. ESEC/FSE*, 2015, pp. 1–12.

[6] J. M. Franco, F. Correia, R. Barbosa, M. Zenha-Rela, B. Schmerl, and D. Garlan, "Improving self-adaptation planning through software architecture-based stochastic modeling," *J. Syst. Softw.*, vol. 115, pp. 42–60, 2016.

[7] A. Filieri et al., "Control strategies for self-adaptive software systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 11, no. 4, 2017, article 24.

[8] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building more robust cloud applications," in *Proc. ICSE*, 2014, pp. 700–711.

[9] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *FOSE*, 2007, pp. 259–268.

[10] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *Proc. CCGrid*, 2017, pp. 64–73.

[11] R. R. Filho and B. Porter, "Defining emergent software using continuous self-assembly, perception, and learning," *ACM Trans. Auton. Adapt. Syst.*, vol. 12, no. 3, pp. 16:1–16:25, 2017.

[12] Z. A. Mann, "Resource optimization across the cloud stack," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 1, pp. 169–182, 2018.

[13] ——, "Modeling the virtual machine allocation problem," in *Proc. Int. Conf. on Mathematical Methods, Mathematical Models and Simulation in Science and Engineering*, 2015, pp. 102–106.

[14] E. Ahvar, S. Ahvar, Z. A. Mann, N. Crespi, J. Garcia-Alfaro, and R. Glitho, "CACEV: a cost and carbon emission-efficient virtual machine placement method for green distributed clouds," in *Proc. 13th IEEE Int. Conf. on Services Computing*, 2016, pp. 275–282.

[15] M. Hinchey, S. Park, and K. Schmid, "Building dynamic software product lines," *IEEE Computer*, vol. 45, no. 10, pp. 22–26, 2012.

[16] L. Chen and H. Shen, "Consolidating complementary VMs with spatial/temporal-awareness in cloud datacenters," in *Proc. IEEE INFOCOM*, 2014, pp. 1033–1041.

[17] Z. A. Mann, "Approximability of virtual machine allocation: much harder than bin packing," in *Proc. 9th Hungarian-Japanese Symp. Discrete Math. Appl.*, 2015, pp. 21–30.

[18] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software Pract. Exper.*, vol. 41, no. 1, pp. 23–50, 2011.

[19] Z. Á. Mann, "Cloud simulators in the implementation and evaluation of virtual machine placement algorithms," *Software: Practice and Experience*, vol. 48, no. 7, pp. 1368–1389, 2018.

[20] ——, "Two are better than one: An algorithm portfolio approach to cloud resource management," in *Proc. ESOCC*, 2017, pp. 93–108.

[21] C. Krupitzer, F. M. Roth, M. Pfannemüller, and C. Becker, "Comparison of approaches for self-improvement in self-adaptive systems," in *Proc. ICAC*, 2016, pp. 308–314.

[22] M. U. Iftikhar and D. Weyns, "ActivFORMS: Active formal models for self-adaptation," in *Proc. SEAMS*, 2014, pp. 125–134.
[23] S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu, Eds., *Self-Aware Computing Systems*. Springer International Publishing, 2017.
[24] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proc. ICSE*, 2014, pp. 299–310.
[25] K. Åström and B. Wittenmark, *Adaptive control*. Courier Corp., 2013.
[26] J. Grohmann, N. Herbst, S. Spinner, and S. Kounev, "Self-tuning resource demand estimation," in *Proc. ICAC*, 2017, pp. 21–26.