# A decision-theoretic approach for prioritizing maintenance activities in organic computing systems

**Markus Görlich-Bucher, Michael Heider, Tobias Ciemala, Jörg Hähner**

**Angaben zur Veröffentlichung / Publication details:**

Görlich-Bucher, Markus, Michael Heider, Tobias Ciemala, and Jörg Hähner. 2023. "A decision-theoretic approach for prioritizing maintenance activities in organic computing systems." In *Architecture of Computing Systems: 36th International Conference, ARCS 2023, Athens, Greece, June 13–15, 2023, proceedings*, edited by Georgios Goumas, Sven Tomforde, Jürgen Brehm, Stefan Wildermann, and Thilo Pionteck, 37–47. Cham: Springer. https://doi.org/10.1007/978-3-031-42785-5_3.



# A Decision-Theoretic Approach for Prioritizing Maintenance Activities in Organic Computing Systems

Markus Görlich-Bucher(✉), Michael Heider, Tobias Ciemala, and Jörg Hähner

Organic Computing Group, University of Augsburg, Augsburg, Germany
{markus.goerlich-bucher,michael.heider,tobias.ciemala,
joerg.haehner}@uni-a.de
https://www.uni-augsburg.de/en/fakultaet/fai/informatik/prof/oc/

**Abstract.** Organic Computing systems intended to solve real-world problems are usually equipped with various kinds of sensors and actuators in order to be able to interact with their surrounding environment. As any kind of physical hardware component, such sensors and actuators will fail after a usually unknown amount of time. Besides the obvious task of identifying or predicting hardware failures, an Organic Computing system will furthermore be responsible to assess if it is still able to function after a component breaks, as well as to plan maintenance or repair actions, which will most likely involve human repair workers. Within this work, three different approaches on how to prioritize such maintenance actions within the scope of an Organic Computing system are presented and evaluated.

**Keywords:** Organic Computing · Predictive Maintenance · Decision Theory

## 1 Introduction

The interdisciplinary research domain of *Organic Computing* (OC) [8] is concerned with solving the increasing complexity in information- and communication technology by allowing systems to freely adapt and organize themselves. In order to interact with their surrounding environment, OC-based systems are expected to include various kinds of *sensors* and *actuators*. Accordingly, a significant amount of OC research conducted over the past two decades focused on building systems and architectures that are inherently focused on dealing with various real-world scenarios. [9] Hereby, a prominent aspect of OC systems is their ability to remain *robust* with respect to various kinds of *disturbances* that may happen within the scope of the OC system. In terms of OC, the idea of robustness is to remain functioning at a desired level of system performance while a disturbance occurs. Strongly related is the concept of *self-healing*, which describes a systems capability of *healing* itself from disturbances or system failures in an abstract manner [9]. As for now, most OC research concerned with

robustness and self-healing focuses on dealing with software-sided disturbances, as discussed in [4]. Although quite relevant due to the real-world focus, a rarely discussed aspect in OC research is robustness against hardware-related disturbances, such as component failures or breakdowns. Besides the quite obvious challenges of identifying or predicting hardware-related disturbances, a notable problem lies in actually *solving* such a disturbance: As for now, one can expect that sensors and actuators (or summarized: *components*) of an OC system settled in a real-world context need to be repaired or maintained by some sort of human repair worker. Furthermore, until the human repair worker is able to maintain or replace a broken component, the overall robustness of the system may be endangered: In contrast to software-sided disturbances, which may be compensated by e.g. starting another instance of a software module, a hardware-related persists until being repaired. If the overall system configuration does not provide some sort of hardware redundancy, the system may not be able to remain robust. Accordingly, it is of interest to both *assess* how the system will be perform in the future considering known (that is: current) or potential (that is: predicted) broken components, as well as to *plan* or *prioritize* certain repair or maintenance actions involving said human repair workers. Within the scope of this paper, we present a brief example on how to plan such maintenance actions. We evaluate our approach using an example of an *organic production line* as well as a datacenter.

The remainder of this paper is structured as follows. In Sect. 2, we give a brief overview on existing work on the topic from the field of OC, as well as relevant work from other research domains. Afterwards, we describe some assumptions and prerequisites as well as the overall system model for our concept in Sect. 3. We describe our methodology in detail in Sect. 4 before providing evaluating and discussing our approach in Sect. 5. We conclude with a brief outlook on possible future work in Sect. 6.

## 2 Related Work

A brief introduction of the original concept of robustness, as it is considered in OC can be found in [8]. A more contemporary approach focusing on actually measuring robustness is given in [13]. As already outlined in the introduction, existing OC research on self-healing mostly focuses on software-sided disturbances (cf. [11,12]). The few work concerned with self-healing of hardware-related disturbances mostly focuses on compensating breakdowns rather than repairing or maintaining them, as for example [7]. A more detailed introduction on the general problem of hardware-sided disturbances in OC can be found in [3] as well as [4].

A research domain concerned with quite similar problems as those discussed in this work is the broader field of *Predictive Maintenance* (PdM). PdM focuses both on aspects of predicting breakdowns in various kinds of machinery, usually using some sort of machine learning algorithm or statistical methods, as well as the aspect of planning actual maintenance tasks. A brief introduction on the

general topic can be found in [2]. Some notable work on planning maintenance actions can be found in [6,14] as well as [5].

A quite relevant aspect of maintenance planning work from the PdM domain is that PdM usually focuses on long-term planning of maintenance actions, as comes clear when considering e.g. [5]. This does not come to much of a surprise, as plenty existing PdM work is motivated by various subdomains of mechanical engineering. Shutting down whole production plants with static manufacturing processes and complex machinery, as for example in the automotive sector, may require a significant amount of planning in advance. OC systems, as we expect on the other hand, are affiliated with a more agile and flexible mode of operation. Accordingly, it is more of interest to provide a more short-hand and flexible way of planning maintenance actions.

## 3 Prerequisites

In the following section, various prerequisites necessary for the remainder of this paper are discussed. First of all, we give a brief overview on the overall system model based on the existing *Multi Level Observer Controller*-architecture. Afterwards, the actual problem statement of this paper is discussed thoroughly.

We assume an Organic Computing system $S$ based on the *Multi-Level Observer Controller*-architecture (MLOC) [8]. In brief words, a *System under Observation and Control* (SuOC), which kind of acts as a representation for the real world aspect of $S$, is equipped with a *Control Mechanism* (CM) incorporating various *Observers* as well as a *Controllers* situated in several *Layers*. Layer 0 refers to the productive part, accordingly, to the SuOC. Layer 1 can be summarized as the reactive part of the whole architecture and is responsible for taking immediate action depending on the current system state. Layer 2 is responsible for gaining new knowledge (e.g. on previously unknown system states), probably by utilizing a simulation of the SuOC. Finally, Layer 3 consists various kinds of collaboration mechanisms, such as communication with other MLOC instances as well is responsible providing monitoring or goal management capabilities for external users. We refer to [9] for a broader introduction on the overall topic.

In a more formal way, the OC system $S$ is affiliated with a *state space* of possible system states $Z$. For each observed state $z(t)$ at timestep t, $S$ reacts with an (preferably suitable) action $a$. A disturbance $\theta$ at timestep $t$ is defined as a change of a system state $z(t)$ to some other state $\theta(z(t))$. Taking the quantification methodology from [13] into account, each $z$ is furthermore associated with a utility measure $U$, which can be used to asses the overall performance of $S$. If a disturbance $\theta$ is able to change $U$ below some predefined threshold, $U_{acc}$, $S$ performs no longer in an acceptable manner. If the system is not able to recover from this drop in utility, it cannot be considered as robust.

Considering the actual composition of $S$, we assume that the SuOC consists of several *components* $c \in C$. Each component is associated with an unknown degree of wear, therefore, will presumably *fail* some time in the future. We assume that $S$ is able to both identify currently broken components (e.g. by

some sort of health signal) as well as to, affiliated with some uncertainty, predict future breakdowns of components (e.g. by using some sort of prediction algorithm trained on previous sensor readings from failing components). Accordingly, each $c \in C$ is associated with a specific measurement of its current wear state, that is, if it is functioning or broken. We call this measurement *integrity*. In a more formal way, the integrity measure for a component $c_j$ can be defined as $c_j^i \rightarrow \{0, 1\}$ with 1 determining a functional and 0 determining a broken component. The uncertain prediction of a components functionality in a discrete, known time horizon can be defined as $c_j^p \rightarrow \{0, 1\}$ with 0 determining a predicted breakdown and 1 determining a predicted functional component. This prediction is furthermore associated with a *confidence* measure $c_j^c \rightarrow [0, 1]$ determining how reliable the prediction is. In order to reflect the physical state of the system in the state space of $S$, we define $Z_C \subset Z$ as a state space containing limited system state descriptions only concerned with describing the integrity measures of all $c \in C$, that is, information on identified or predicted breakdowns. We furthermore assume that $S$ entirely knows the cost function $Z_C \rightarrow U$, meaning that $S$ is able to evaluate how its overall utility is affected by the integrity of its components.

From an architectural point of view, the methodology for identifying and predicting breakdowns would be found in the layer 1 observer, while the algorithms for prioritizing maintenance actions would be executed within the layer 1 controller. The component for calculating the cost function for evaluating the influence of a broken component on the system utility, however, would be located in the layer 2 controller, as one can expect that it makes use of the corresponding simulation component in order to learn the cost function. Taking these known measurements into account, it is of interest to plan necessary maintenance actions such that the overall system performance in terms of utility is ensured. Three suitable approaches are presented in the following section.

## 4 Methodology

At each discrete timestep, $S$ is expected to assign a *priority* $\omega$ to each $c \in |C|$, taking the current system state as well as predictions of future breakdowns into consideration. As mentioned earlier, it is expected that $S$ has access to a perfect cost function, allowing it to assess how the system will perform in case of a breakdown, presumably by utilization of a simulation on MLOC layer 2. The approach presented in this paper focuses on prioritizing maintenance activities for a horizon of one discrete timestep. However, the concept should be applicable to more extensive time horizons as well.

Two of the three algorithms presented in this work are based on concepts from the broader field of decision theory. In brief words, the general idea of decision theory is to formalize the problem of making decisions under uncertainty. In order to do so, decision theory-based methodologies assess potential outcomes based on their probability, combined with a utility function, in order to calculate an expected utility for a discrete timestep. We refer to [10] for a broader introduction

on the topic. Taking the formalization of the previous chapter into account, the *expected utility* $E_U(z)$ for a system state $z$ is calculated as follows:

$$E_U(z) = \sum_{i=0}^{|Z_C|} p(z_i) * U(z_i) \tag{1}$$

Here, a (physical) system state $z_i$ consists of the integrity measures of all $c \in C$. Accordingly, the expected utility for a given state is calculated using the current occurrence probabilities of all possible system states as well as their corresponding system utilities. The occurrence probabilities are calculated using the breakdown predictions as well as the confidence scores for all components as follows:

$$p(z_i) = \prod^{c \in C} \begin{cases} c_j^c & \text{if } c_j^i = 0 \\ 1 - c_j^c & \text{if } c_j^i = 1 \end{cases} \tag{2}$$

In simple words, the expected utility describes the expectation value for the system utility under consideration of all possible system states. The probability of a single system state is therefore calculated using the prediction confidences under assuming the specific integrity values taken from this very system state. From a practical point of view, calculating $E_U(z)$ would allow $S$ to assess how its utility would presumably develop under consideration of its current state, as well as its current predictions. However, in general, calculating $E_U(z)$ is quite expensive due to the combinatorial complexity of taking all conceivable system states into account.

The *priority* $\omega$ of performing a maintenance action $a_j$ on component $c_j$ can be regarded as directly proportional to improvement of the expected utility after performing said maintenance action. More formally, $\omega$ can be defined as follows:

$$\omega(a_j) \propto \Delta E_U(a_j) \tag{3}$$

Two decision theory-based methods implementing $\Delta E_U$ are discussed in the following. The simple stochastic planing algorithm, as explained in the next section, focuses on a simplified approach by reducing the complexity coming from taking all possible system states into account. The complex stochastic planing algorithm, which is explained afterwards, actually utilizes $E_U(z)$ for planing maintenance operations.

### 4.1 Simple Stochastic Planing

Within the simple stochastic planing algorithm, $\Delta E_U$ is calculated using a *simplified* expected utility function. This function does not take all possible system states and their probabilities into account, but focuses only on the prediction confidence associated with the component $c_j$. The function can be defined as follows:

$$E'_U(z, c_j) = U(z|c_j^i = 0) * c_j^c + U(z|c_j^i = 1) * (1 - c_j^c) \tag{4}$$

Using $E'_U$, $\Delta E_U$ can be calculated as follows:

$$\Delta E_U(a_j) = E'_U(z, c_j|a_j) - E'_U(z, c_j) \tag{5}$$

Hereby, $E'_U(z, c_j|a_j$ refers to the simplified expected utility function under the assumption that a maintenance action was performed on $c_j$ just as $E'_U$ is calculated. In these cases, it is assumed that $c_j^c = 0$, therefore, that probability of $c_j$ failing equals 0. Briefly, the calculation of $\omega$ for a component $c_j$ is based on the difference between expected utility if $c_j$ is maintained and the expected utility if it is not maintained.

### 4.2 Complex Stochastic Planing

The complex stochastic planing algorithm works quite similar to the simple stochastic planing algorithm. The major difference is that the both the simplified expected utility, as well as the actual expected utility are utilized for the calculation of the priorities:

$$\Delta E_U(a_j) = E'_U(z, c_j|c_j^i = 1) - E_U(z) \tag{6}$$

In simple words, using the original expected utility instead of the simplified version makes sure that the calculation considers interactions between components. Accordingly, other possible breakdowns and their probabilities are taken into account. As mentioned before, the calculation of $E_U(z)$ may be quite expensive, depending on $|C|$.

At this point, one could argue that it is unnecessary to calculate $E_U(z)$, as it is subtracted in each priority calculation. Accordingly, the sequence of the priorities remains the same, when it is left out of the equation. This is indeed correct, however, as it may be of interest to assess if the acceptance or survival boundaries of the system might be violated, the calculation might be necessary, depending on the actual application scenario.

### 4.3 Naive Planing

In order to provide some sort of most simple ground truth, a naive, greedy planing algorithm is implemented. The naive algorithm is defined as follows:

$$\omega(a_j) = p(z_i) \tag{7}$$

Accordingly, following Eq. 2, a component currently broken is prioritized with 1 (as a maximum priority), while functioning components are prioritized based on their predictions and prediction confidences.

## 5 Evaluation

We evaluate our approach using two scenarios of different complexity. In the following, both scenarios are described briefly. Afterwards, the results of the evaluation are shown and discussed.

### 5.1 Datacenter Scenario

The idea of the datacenter scenario is to simulate the behaviour of hard drives in RAID-arrays, more specifically, RAID5. Accordingly, the components in this scenario are hard drives. Each RAID-array consists of three hard drives. The utility of a single RAID-array is the reading speed of the combined hard drives. Accordingly, if one drive fails, the reading speed decreases. If two drives fail, the RAID is no longer functional, therefore, its reading speed decreases to 0. In order to simulate a realistic behaviour, the runtimes and breakdowns of the hard drives are calculated based on datasets from the hosting company backblaze[1]. There exists work on predicting hard drive failures using the S.M.A.R.T. logging data provided by a hard drive's controller (e.g. [1,16]).

The hard drives within the datacenter are used as components within this scenario, therefore, additional components like RAID-controllers are not considered. The scenario consists of 200 RAID-arrays with a runtime of 365 discrete timesteps. Here, one timestep equals one day.

### 5.2 Organic Production Line Scenario

In order to emphasize the inherent real world aspect of OC, our Organic Production Line (OPC) scenario is inspired by a pulley factory layout as shown in [15]. The overall layout of the production line is shown in Fig. 1 and briefly described in the following. An *Entry* describes the entry point for raw material necessary for the production line. As can be seen, three different types of raw material are induced into the OPC. A *Buffer* describes some sort of buffering appliance where semi-processed work pieces or, later on, fully processed products are stored until their further processing. Various buffers are used for *splitting* the production line—for example, when two redundant machines work in parallel—as well as for *combining* processing lines—accordingly, when two machines in parallel produce the same work piece. A single *Exit* describes the end of the production line, therefore, the final execution step where processed units leave the OPC. Finally, various kinds of *Processing Machinery* (shown by the rectangles) are involved in processing the work pieces from step to step.

From an OC point of view, the processing machinery can be regarded as all components existing in $C$. For the scope of our evaluation, we assume that the buffers, entries and exits, as well as other equipment not shown in the layout (e.g. conveyor belts) do not feature sensors/actuators or do not belong to the scope of $S$. $U$ can be regarded as the number of assembled units per time leaving the exit, while the maximum number of produced units per discrete timestep is $U_{max} = 100$. Intuitively, the minimal number of produced units is $U_{min} = 0$. The scenario consists of 25 equal OPCs and 500 discrete timesteps. Here, one discrete timestep equals one hour, therefore, the overall simulation covers about

[1] https://www.backblaze.com/b2/hard-drive-test-data.html.

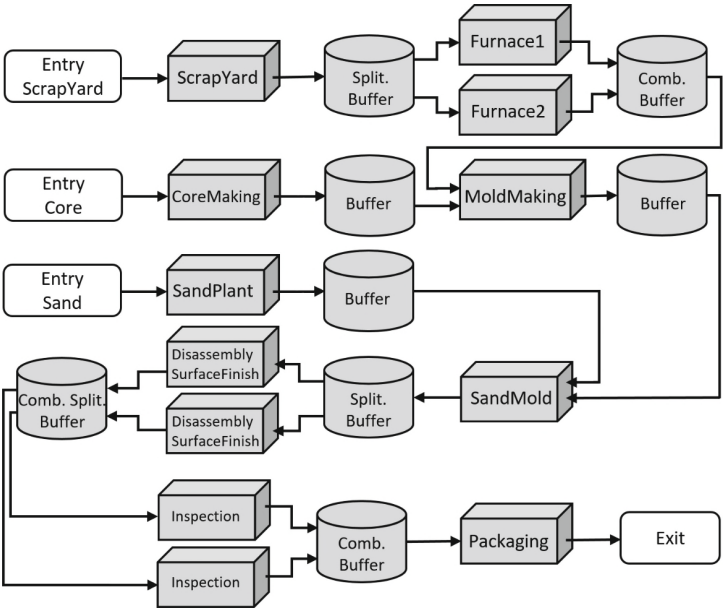


**Fig. 1.** Layout of the OPC scenario based on [15]

three weeks. Runtimes and breakdown behaviour of the machines in the scenario is based on the Azure AI Predictive Maintenance Dataset[2].

### 5.3 Results

As mentioned before, a perfect cost function is assumed for both scenarios. Besides, for predicting upcoming disturbances, we assume a maximum prediction horizon of three discrete timesteps. In order to simulate uncertainty, the predictions for the longest horizon are correct with a chance of 50%, while those with the shortest prediction horizon (that is: next discrete timestep) are correct with a chance of 90%. Additionally, the probabilities are altered using a normally distributed spread of 10%. 30 runs with different random seeds were conducted for each experiment. For now, no survival space boundary was incorporated.

The cumulated total utility per repetition as well as the average utility per timestep for the datacenter scenario are shown in Table 1. In order to check the results on significance, one-sided t-Tests with $\alpha = 0.05$ were performed on each pair of algorithms. A previously performed Shapiro-Wilk-Test showed that the assumption of a normal distribution cannot be rejected for all data. In total, the complex stochastic algorithm performed significantly superior to all other algorithms with $p = 9.48 * 10^{-15}$ for the naive algorithm as well as $p = 1.02 * 10^{-17}$ for the simple stochastic algorithm.

[2] https://www.kaggle.com/datasets/arnabbiswas1/microsoft-azure-predictive-maintenance.

**Table 1.** Cumulated utility per repetition and average utility per timestep for the datacenter scenario

| Algorithm | Cumulated utility | Average utility |
|---|---|---|
| Simple stochastic | $6877133.33 \pm 104284.60$ | $94.21 \pm 1.43$ |
| Complex stochastic | $6879315.56 \pm 103708.81$ | $94.24 \pm 1.42$ |
| Naive | $6810626.67 \pm 129186.03$ | $93.29 \pm 1.77$ |

The cumulated total utility per repetition as well as the average utility per timestep for the OPC scenario are shown in Table 2. Again, a previously performed Shapiro-Wilk-Test showed that the assumption of a normal distribution cannot be rejected for all data. Therefore, this scenario was also tested using one-sided t-Tests with $\alpha = 0.05$. Hereby, the complex stochastic algorithm also performed significantly superior to the other algorithms with $p = 1.66 * 10^{-18}$ for the naive algorithm and $p = 2.36e * 10^{-5}$ for the simple stochastic algorithm.

**Table 2.** Cumulated utility per repetition and average utility per timestep for the OPC scenario

| Algorithm | Cumulated utility | Average utility |
|---|---|---|
| Simple stochastic | $914905.52 \pm 26934.46$ | $73.19 \pm 2.15$ |
| Complex stochastic | $930305.83 \pm 27585.01$ | $74.42 \pm 2.21$ |
| Naive | $869469.17 \pm 26718.52$ | $69.56 \pm 2.14$ |

### 5.4 Discussion

For both scenarios, the decision theory-based algorithms show a significantly better performance compared to the naive approach.

While the results for the datacenter scenario appear quite similar, the production line scenario shows a quite notable difference between the three algorithms. This might be due to the fact that the production line scenario incorporates more interdependencies between the various components: For the datacenter, only three components, that is, hard drives per raid show direct dependencies, while one production line consists of twelve dependent components. Accordingly, one single breakdown in a production line has a bigger influence on the whole utility as it would have in the datacenter scenario. This results in slightly worse prioritizations—as, for example, the naive algorithm deciding greedily based on the breakdown probability—having a bigger impact on the overall utility. This aspect could also explain the noticeably better performance of the complex stochastic method (when compared to the simple stochastic method), as it takes the potential influences of other broken components into account. Accordingly,

consideration of all possible futures system states within the complex stochastic method appears to be beneficial for systems with a higher degree of dependencies.

## 6 Conclusion and Outlook

In this work, two decision-theoretic approaches on prioritizing maintenance actions in organic computing systems were presented and evaluated alongside a naive method. It was shown that the decision-theoretic approach is indeed able to perform significantly better than the naive approach. Also, it was shown that the presented complex variant may yield better results in scenarios with a higher amount of dependencies between components.

In general, the presented methodologies base on rather simplified assumptions on how a real-world scenario may look like. There are various aspects not considered yet, such as parallel maintenance actions or different maintenance costs. Another relevant aspect, especially considering the OPC scenario, would be the explicit consideration of breakdowns affecting the utility in a delayed manner. Finally, within the scope of this work, a perfect utility function as well as rather simplified assumptions on the behaviour of the overall system were assumed, although is questionable if these assumptions could made for more complex OC scenarios in real world settings. This could incorporate the necessity to actually learn a suitable (probably imperfect) utility function, as well as to deal with unknown system states, changing probabilities for their occurrence, as well as imperfect or inaccurate predictions.

## References

1. Aussel, N., Jaulin, S., Gandon, G., Petetin, Y., Fazli, E., Chabridon, S.: Predictive models of hard drive failures based on operational data. In: 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 619–625. IEEE (2017)
2. Carvalho, T.P., Soares, F.A., Vita, R., Francisco, R.D.P., Basto, J.P., Alcalá, S.G.: A systematic literature review of machine learning methods applied to predictive maintenance. Comput. Ind. Eng. **137**, 106024 (2019)
3. Görlich, M., Stein, A., Hähner, J.: Towards physical disturbance robustness in organic computing systems using MOMDPs. In: ARCS Workshop 2019; 32nd International Conference on Architecture of Computing Systems, pp. 1–4. VDE (2019)
4. Görlich-Bucher, M.: Dealing with hardware-related disturbances in organic computing systems. In: INFORMATIK 2019: 50 Jahre Gesellschaft für Informatik-Informatik für Gesellschaft (Workshop-Beiträge). Gesellschaft für Informatik eV (2019)
5. Hardt, F., Kotyrba, M., Volna, E., Jarusek, R.: Innovative approach to preventive maintenance of production equipment based on a modified TPM methodology for industry 4.0. Appl. Sci. **11**(15), 6953 (2021)
6. Ji, B., et al.: A component selection method for prioritized predictive maintenance. In: Lödding, H., Riedel, R., Thoben, K.-D., von Cieminski, G., Kiritsis, D. (eds.) APMS 2017. IAICT, vol. 513, pp. 433–440. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66923-6_51

7. Maehle, E., et al.: Application of the organic robot control architecture ORCA to the six-legged walking robot OSCAR. In: Müller-Schloer, C., Schmeck, H., Ungerer, T. (eds.) Organic Computing-A Paradigm Shift for Complex Systems, pp. 517–530. Springer, Basel (2011). https://doi.org/10.1007/978-3-0348-0130-0_34
8. Müller-Schloer, C., Schmeck, H., Ungerer, T.: Organic Computing-A Paradigm Shift for Complex Systems. Springer, Basel (2011). https://doi.org/10.1007/978-3-0348-0130-0
9. Müller-Schloer, C., Tomforde, S.: Organic Computing – Technical Systems for Survival in the Real World. AS. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68477-2
10. Pratt, J.W., Raiffa, H., Schlaifer, R., et al.: Introduction to Statistical Decision Theory. MIT Press, Cambridge (1995)
11. Satzger, B., Pietzowski, A., Trumler, W., Ungerer, T.: Variations and evaluations of an adaptive accrual failure detector to enable self-healing properties in distributed systems. In: Lukowicz, P., Thiele, L., Tröster, G. (eds.) ARCS 2007. LNCS, vol. 4415, pp. 171–184. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71270-1_13
12. Schmitt, J., Roth, M., Kiefhaber, R., Kluge, F., Ungerer, T.: Using an automated planner to control an organic middleware. In: 2011 IEEE Fifth International Conference on Self-Adaptive and Self-Organizing Systems, pp. 71–78. IEEE (2011)
13. Tomforde, S., Kantert, J., Müller-Schloer, C., Bödelt, S., Sick, B.: Comparing the effects of disturbances in self-adaptive systems - a generalised approach for the quantification of robustness. In: Nguyen, N.T., Kowalczyk, R., van den Herik, J., Rocha, A.P., Filipe, J. (eds.) Transactions on Computational Collective Intelligence XXVIII. LNCS, vol. 10780, pp. 193–220. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78301-7_9
14. van Horenbeek, A., Pintelon, L.: A dynamic predictive maintenance policy for complex multi-component systems. Reliab. Eng. Syst. Saf. **120**, 39–50 (2013). https://doi.org/10.1016/j.ress.2013.02.029
15. Watanapa, A., Kajondecha, P., Duangpitakwong, P., Wiyaratn, W.: Analysis plant layout design for effective production. In: Proceeding of the International Multi Conference of Engineers and Computer Scientists, vol. 2, pp. 543–559 (2011)
16. Zhu, Y., Wu, P.H.J., Liu, F., Kanagavelu, R.: Disk failure prediction for Software-Defined Data Centre (SDDC). In: 2021 IEEE International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, International Conference on Cloud and Big Data Computing, International Conference on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), pp. 264–268. IEEE (2021)