Proceedings of the 33rd European Safety and Reliability Conference (ESREL 2023)
Edited by Mário P. Brito, Terje Aven, Piero Baraldi, Marko Čepin and Enrico Zio
©2023 ESREL2023 Organizers. *Published by* Research Publishing, Singapore.
doi: 10.3850/978-981-18-8071-1_P122-cd

Towards Verification of Self-Healing for Autonomous Vehicles

Timo Frederik Horeis and Rhea C. Rinaldo
Institute for Quality and Reliability Management (IQZ GmbH), Hamburg, Germany.
E-mail: horeis@iqz-wuppertal.de, rhea.rinaldo@iqz-wuppertal.de

In the scientific community, "Organic Computing (OC)" offers a promising approach to designing and developing highly reliable and cost-efficient systems, one of the main challenges in developing autonomous vehicles. Thereby, OC proposes the implementation of diverse self*-properties to make the system act self-aware and autonomously throughout environment and requirement changes. These properties include, a.o., the self-configuration of the system, self-healing from erroneous and corrupted states, and self-protecting against security attacks carried out by the system independent from human commands. Regarding the insurance of safety, reliability and security, mainly self-healing is introducing new possibilities for complex, cost-intense systems. Self-healing systems can detect, diagnose and repair failures in a self-aware manner, increasing the robustness and operation time without affording maintenance performed by humans. The emerging benefit lies in enhancing the system's safety and security parameters while keeping costs and resources reasonable.
Current research focuses mainly on architectures and functional implementations of self*-properties. However, to our knowledge, modeling and verification approaches for the properties' impact still need to be developed. This is key for OC being accepted beyond the scientific scope. Therefore, this paper performs a literature study on self-healing and defines its core concept in terms of a generic modeling. This modeling builds the basis for a quantitative verification of self-healing. To show its effectiveness it is implemented in an existing assessment tool, "ERIS", and an example application is presented.

Keywords: Organic Computing, Verification, Self-healing, Self-protecting, Safety, Security.

1. Introduction

In the last decade, the drive towards systems with a higher level of automation has been drastically increasing and concomitantly has the constant growth in the amount and novelty of tasks. This trend towards higher levels of automation leads to an increasing complexity and connectivity of the system. As a consequence, current design techniques are reaching their limits (Tomforde et al. (2014)). Thus in the future, the application of classical top-down approaches for the design of highly-automated vehicles that rely on breaking down the specification into single elements, making knowledge of all system states at all times mandatory, will no longer be applicable. In order to adjust to the degree of complexity and connectivity, new design techniques have to be introduced.

The concept of Organic Computing (OC) offers a promising solution. OC chases the idea of including biological concepts in currently strict and inflexible technical systems. Thereby, the goal is to make the technical system automatically, in a self-aware manner, adapt to new situations and environments as a reaction to external or internal events. The system's adaption possibilities include both; the system structure (components and connections) and the system behavior realized by implementing different self*-properties, e.g., based on machine learning techniques. With the system taking out specific tasks on its own, it does not rely on design time specifications for these anymore, and a playground for autonomously finding and performing problem solutions to fulfill the desired goal is prepared. This idea leads to a significant paradigm shift: former design time decisions are now made at run-time with the decision-making responsibility being transferred from the engineer to the system itself (see also Tomforde (2011)). In addition to a reduced design time, OC properties can benefit a system in many ways, ranging from economic and financial reasons to the ones of functional safety and cybersecurity. Particularly in the field of safety and cybersecurity in the automotive domain, OC offers an excellent opportunity to overcome the dilemma of designing affordable vehicles with sufficient safety and security solutions (Güdemann et al., 2006; Horeis et al., 2022). In this way, OC can be used to achieve robustness against failures without introducing additional components and thus keeping the costs lower compared to traditional design concepts.

However, while the first ideas of OC and its initiative date back to the early 2000s (see Müller-Schloer et al. (2004)), first implementations in the automotive domain have only been proposed in the last decade (see e.g. Schleiss et al. (2014)). One reason is that OC implementations also pose new challenges to the system's safety by introducing new kinds of failures, adding uncertainties, and possibly increasing the reaction time, as well as the interest and risk for attacks (Müller-Schloer et al. (2004)). Further, the applied risk assessment techniques must be capable of considering these effects to represent the system accurately. To conquer these challenges and ensure the acceptance of OC implementations in the automotive domain, modeling and verification techniques capable of considering self*-properties become mandatory. Contributions to this are, however, sparse. First, efforts were made by Ehrig et al. (2010), who propose a way to model and formalize self-healing systems by a tool-supported static verification technique based on graph transformation. Nafz et al. (2010) are making use of compositional verification based on rely/guarantee specifications to verify self-organization and the system's functional behavior separately. Other approaches are verifying the results at run-time, e.g., via online verification like Stahl (2022) or verified result checking as Fischer et al. (2011). With this paper, we aim to push this topic further by developing a modeling technique for self-healing and implementing it in a pre-developed verification tool. We start off by identifying the main characteristics of OC, more specifically self-healing, in terms of a short literature research. Next, a modeling approach of the identified characteristics is proposed, which is then implemented into a pre-developed modeling tool and an exemplary analysis is performed. Lastly a conclusion and outlook of the findings and limitations is given.

2. Organic Computing

Organic Computing (OC) introduces several self*-properties to a system to make it adaptable to new, undefined situations and tasks. To the best of our knowledge, no unique definition of the term OC and the accompanying self*-properties exists. For this paper, we follow the definitions given by Tomforde et al. (2017), who list self-configuration, self-organization, self-integration, self-management, self-healing, self-protecting, self-stabilizing, self-improving, and self-explaining. While it is essential to consider all OC properties of a system in the big picture, for reasons of space, we focus on self-healing in this paper. In the subsequent Section 2.1 a definition of self-healing is given and its characteristics on different implementation levels is expounded.

2.1. *Self-Healing*

Self-healing describes the capability of the system to automatically heal from erroneous states or behaviors caused by failures or attacks. It characterizes by performing repair and recovery actions entirely without human intervention. Failures and attacks can occur on different levels, e.g., hardware faults v.s. software errors. Consequently, healing techniques are deployed on different levels, usually distinguished by system level and component level (consisting of application level and hardware level). On hardware level, classical techniques are based on voter systems such as Dual or Triple Module Redundancy, but also more revolutionary techniques such as Evolvable Hardware (EHW) or Embryonic Hardware (EmHW) are being researched (for more information see Khalil et al. (2019)).

Regarding software a vast amount of approaches exist. Monperrus (2018) provides an exhaustive overview on automatic repair, categorizing behavioral and state repair. Behavioral repair includes, among others, approaches that prevent software crashes by making the code safer through automatic patching like Azim et al. (2014); Dennis et al. (2006). Classical state repairs comprise the restart (reboot) of the affected component to reclaim stale resources, clean up the corrupted state and fix Heisenbugs[a] (see also Candea and Fox (2001)), the rollback to a previous operational state by making use of checkpoints and snapshots, or the entire switch to a different program such as the idea of n-version programming. Further, state repairs concern reconfiguration and can be used for self-healing by, e.g., moving software applications between multiple computing nodes (Kain et al. (2020)), input modification and error virtualization. In Keromytis (2007), approaches focusing on recovering from attacks are presented. For example, DIRA by Smirnov and Chiueh (2005) automatically detects and recovers (control-hijacking) buffer overflow attacks by preventing the identified attack from propagating and rolling back to a safe state if necessary.

On system level, self-healing approaches focus on identifying and correcting poorly performing processes. For instance, in the SafeAdapt project (Schleiss et al. (2014)), a software core performing self-configuration to achieve an effective fault-tolerant electrical and electronic architecture for autonomous vehicles was developed. Following a similar idea, Kain et al. (2020) developed "FDIRO" as an extension of the well-established Failure Detection Identification and Recovery of the space domain for automotive, with the addition of a self-organization step. The goal of the authors is to achieve higher fault tolerance and availability. Both approaches achieve self-healing through the deployment of other self*-properties.

However, regardless on what level self-healing is concerned, we can observe that the process scheme, visualized in Fig. 1, is identical (see also Psaier and Dustdar (2011). First, some monitoring mechanisms must detect and identify a failure. For various software approaches, such monitoring is an *oracle* that states whether the observed behavior matches a given specification. Then this failure must be located and its cause diagnosed. Once information about the failure is gathered, the system tries to adapt itself by generating possible fixes for the problem. These are tested, and the one that is offering the best target state is deployed.

[a]Bugs that are difficult to reproduce because they often vanish during their investigation

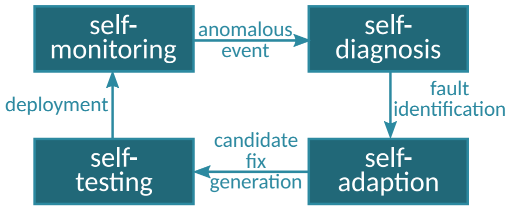

Fig. 1. Self-healing Scheme

3. Modeling Self-Healing

In contrast to the previously introduced approaches, we are interested in modeling the resulting behavior of the implemented self-healing techniques. Therefore, we must include all self-healing steps displayed by Fig. 1 in Section 2.1. Thus we need to define four modeling functions: self-monitoring M, self-diagnosis Δ, self-adaption A and self-testing T. Earlier we saw that self-healing could be performed on different levels. While hardware and software approaches are part of the same component, but system level approaches concern multiple components, we develop two individual approaches for either level. Furthermore, we saw that safety and security repairs differ due to the nature of their cause. For example, Heisenbugs can often be healed by system restarts, while malicious code persists and would require a software rollback. Therefore it is necessary to model them separately.

3.1. *Component Level*

The self-monitoring function M models the success of the implemented self-monitoring technique. Thereby more than one self-monitoring function can be modeled. In this way, we can consider distinct monitoring functions for safety M_{safe} (e.g. representing a Watchdog) and for security M_{sec} (e.g. representing an Intrusion Detection System (IDS)) of the same component. We define $M(t)$ as the probability that the failure or attack is detected at time t. Assuming a random distribution of the detection time with a detection rate θ, we can define a specific $M_k(t)$ as

$$M_k(t) = \begin{cases} 0 & t < t_{failed} \\ 1 - e^{-\theta*(t-t_{failed})} & t \geq t_{failed} \end{cases} \tag{1}$$

whereby t_{failed} is the point in time the failure or attack in the component occurred. In case the behavior of the implemented self-monitoring cannot be described by distribution functions or similar mathematical equations, the following worst-case assumption can be used:

$$M_k(t) = \begin{cases} 0 & t < t_{failed} + t_{max\_M} \\ 1 & t \geq t_{failed} + t_{max\_M} \end{cases} \tag{2}$$

With the worst-case assumption, the duration of self-monitoring always takes time t_{max_M}. Thereby t_{max_M} is defined as the maximum time the self-monitoring procedure is allowed to take. Otherwise, self-monitoring has failed. Similarly, self-diagnosis Δ and self-testing T are modeled. Thereby, we define $\Delta(t)$ as the probability that the cause of the failure or attack is found at time t. By assuming a random distribution of the diagnosis time with a diagnosis rate δ and t_M being the point in time that the self-monitoring detected the failure or attack. A specific $D_k(t)$ can be defined in the same fashion as Eq. (1).

$T(t)$ defines the probability that the component performed the self-testing successfully at time t. Assuming a random distribution of the self-testing time with a testing rate τ and t_A being the point in time that the self-adaption process ended, a specific $T_k(t)$ can also be defined in the same fashion as Eq. (1).

For $\Delta(t)$ and $T(t)$ the worst case assumptions displayed in Eq.(2) can be used accordingly with the maximum duration times t_{max_Δ} and t_{max_T}.

In our opinion the self-adaption step is the most significant one, as it is presumably the step with the highest effort, criticality and time consumption. For self-adaption on component level, we model the self-adaption process by assigning to each repairable application or hardware one or more self-adaption models A_i. Thus, it is possible to model safety and security self-adaption processes individually. In addition, it is also possible to create more than one safety or security self-adaption model per hardware or application. Thereby, the number i of self-adaption models depends on the implemented behavior. Each implemented self-adaption process is represented by its model A_i. The model itself can be represented by a function of time t and a self-adaption rate μ. Thereby, $A_i(t)$ describes the probability that a component that has not been available at time t will be available in the interval $(t, t+dt)$. Assuming a self-adaption process with a random repair behavior, a specific model $A_k(t)$ can be described with an exponential distribution function:

$$A_k(t) = \begin{cases} 0 & t < t_\Delta \\ 1 - e^{-\mu*(t-t_\Delta)} & t \geq t_\Delta \end{cases} \tag{3}$$

whereby t_Δ is the point in time the cause of failure or attack is detected.

In addition to the direct modeling of the failure and attack behavior, e.g., via a failure or attack distribution function $F(t, \lambda)$, it is now possible to also include modeling of the here presented four steps of the self-healing scheme as visualized in Fig. 2.

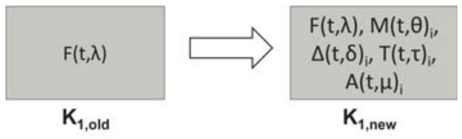

Fig. 2. Self-healing Model Extensions

3.2. *System Level*

For self-monitoring, self-diagnosis and self-testing, we are using a similar modeling idea as on component level in case additional implementation for these steps exist on system level. Otherwise, the self-adaption on system level can also be triggered by self-monitoring or self-diagnosis on component level in case the self-adaption process on component level has failed or is unavailable. For self-adaption on system level, we are proposing two modeling techniques:

(1) Self-adaption triggered by another component
(2) Self-adaption via an adjustment of the system configuration

3.2.1. *Self-Adaption by another Component*

On system level, we have to consider that a self-adaption process can also be triggered by another component, e.g., some superordinate voter or by an over-the-air update. This concept includes additional repair behaviors that have to be modeled. Due to the huge variety, individual modeling solutions or worst-case assumptions should be used, which consider the dependencies between the triggered and triggering components. For example, it must be ensured that the communication between the triggered and triggering components is not corrupted.

3.2.2. *Self-Adaption via System Configuration*

As mentioned in Section 2.1 existing system-level approaches usually employ self-configuration by dynamically changing software instances to available hardware resources. Thereby three main steps can be identified:

- isolation of failed application instance
- determination of new system settings
- deployment of new system setting

These steps are modeled by a probability function, similar to the ones in Section 3.1. Thereby $f_{iso}(t)$ describes the probability at time t that the failed/erroneous application has been successfully isolated. Assuming a random distribution of the isolation time with an isolation rate η, we can define $f_{iso}(t)$. Accordingly, the probability for the determination of the new system setting $f_{det}(t)$ and the probability for the deployment $f_{dep}(t)$ of the new system setting can be defined.

4. Implementation in ERIS

ERIS is a tool that implements a mathematically formalized method for the safety and security assessment of critical, connected systems firstly published in Rinaldo and Hutter (2020), by providing a user interface for modeling, coupled with an evaluation based on probabilistic model checking. It characterizes by a joint modeling of safety and security effects which aims to capture how individual component-specific failures, attacks and their interdependencies impact the system's operational capabilities. Subsequently a brief presentation of ERIS, its implementation of the determined self-healing modeling and an exemplary application is given.

4.1. *Overview*

ERIS takes the general system structure with its design in regard to functional data and communication dependencies, component criticality and redundancy as well as their failure and attack behavior into account and abstracts it as a so-called dependency graph. Thereby components are represented by nodes, with the exception of a special node Env that is modeling the system environment where potential attackers reside. The various interrelationships between nodes are captured by directed links of different types. These links are generalized to express functional data dependencies (Fct) like an actuator relying on sensor data, access and command dependencies (Reach) which may be exploited by an attacker and security guarantees (Sec) reflecting the provision of security measures, e.g., by Hardware Security Modules (HSMs).

To reason about component failures and attacks three health states for nodes are distinguished:

- ok: working as intended
- defective: flawed/irresponsive due to a failure
- corrupted: occupied by an attacker

At start all nodes are considered ok. This state may change indicated by the predominant dependencies in combination with the provided failure and attack specifications in terms of exponentially distributed occurrence rates/probabilities. These are assumed to be determined on the basis of the hardware manufacturer's specification, expert knowledge and/or through detailed risk and vulnerability analyses. Based on the given criticality specification, a definition that determines which components require to be active to provide the core system functionality is derived. To perform a quantitative evaluation, the graph is transformed into a Continuous-Time Markov Chain (CTMC) where a state is a configuration of all node's health states and a transition indicates a single health state change. In the big picture, the evaluation yields the probability that the system reaches an undesirable (non-operational) state, due to a combination of safety failures and security attacks. In automotive, we can use ERIS to determine the worst-case behavior of the abstracted vehicle and further, evaluate safety parameters like its availability and reliability. Ultimately ,these results can be used as proof of legal documents or during the development process, e.g., by comparing different architectures.

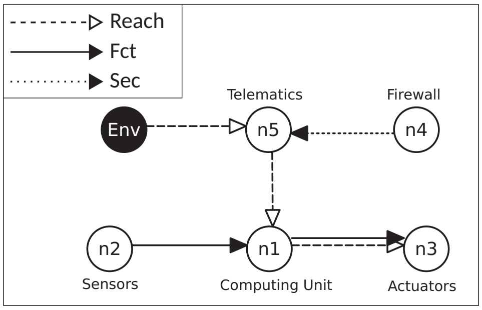

Fig. 3. Autonomous Vehicle Scheme

Figure 3 displays a very abstracted scheme of an autonomous vehicle. A telematics unit is being accessible from the outside and protected by a firewall. It is further having an access relationship to the computing node, the center piece of the self driving system, to, e.g., provide software/parameter updates. Sensors are delivering functionally important data to that computing node, which is processing this data to produce commands for the actuators. For this example a sufficient criticality definition would suggest that the system cannot operate any more once its computing node or the required sensors and actuators cease to operate. In Rinaldo et al. (2021) a detailed ERIS model of an autonomous vehicle can be found.

4.2. *Self-Healing*

On behalf of the presented review on self-healing, we implement the modeling scheme defined in the previous Section 3 in ERIS. Due to ERIS' characteristics, some trade-offs have to be made: The self-testing process is left out completely from the modeling and on component level we only model the performance of the self-adaption process. On system level, self-adaption triggered by other components can be covered, yet self-adaption by adjusting the system configuration cannot, due to ERIS modeling perspective.

Each node n_i is extended by the possibility to define two different exponentially distributed models $A^{n_i}_{safe}(t)$ and $A^{n_i}_{sec}(t)$, which model the self-adaption process of the node after the occurrence of a safety failure $A^{n_i}_{safe}(t)$, respectively a security attack $A^{n_i}_{sec}(t)$. Thereby, the user models the safe-adapting behavior of a defect and a corruption by assigning two different adapting rates μ_{safe} and μ_{sec} to the node. In this way a user-specific choice can be made with respect to the self-adaption process of the represented component. For instance, some components may only implement basic mechanisms like reboots that will not heal a corrupted state. These could be modeled by solely defining a defect self-adaption rate. Others may implement independent mechanisms for safety and security, which are also supported in this way.

On system level we implement the self-adaption process triggered by another component in two ways: (i) Triggered by one or multiple nodes and (ii) triggered by the outside via the Env-node. For both strategies it is essential that the connection to these nodes is not disrupted. This means, there must be a consecutive Reach-path from the triggering node to the triggered node available, where none of the connected nodes is corrupted or defective. For option (i), it is essential that the triggering nodes are operational (ok). With this we model some sort of Watchdog behavior where other components perform the self-monitoring and self-diagnosis and trigger the repair action of the failed/corrupted component. This has the benefit that we can take failures and corruptions of the triggering component itself into account: If this component is failed or corrupted, the repair cannot be triggered. Option (ii) is substantially reflecting an over-the-air update mechanism. Figure 4 shows an example Markov path with recovery of the previous system (Figure 3). First, the firewall application experiences a defect that may be healed. Then the telematics unit is cor-

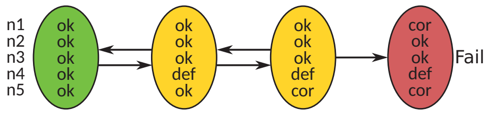

Fig. 4. Markov Path with Self-healing

rupted and if it cannot recover, the computing unit is corrupted eventually, leading to a critical system failure. We can see that in the Markov chain a successful self-healing is simply a transition back to an ok state of the node. For the big picture this means that our system stays potentially longer in non-failed system states and thus the availability of the system can be increased. In our default implementation we do not consider self-adaption in the fail state, because we think that at this point the system has already ceased to operate for an undefined amount of time and we would unnaturally increase its reliability estimation. The reason for that is simple: In the worst-case the vehicle ceases to operate on an active lane, unable to guarantee the safe state.

To show the impact of self-healing we performed an exemplary Markov analysis for reaching the fail state in our pictured system with and without self-healing. Thereby we assumed failure and attack rates in the range of $0\frac{1}{h}$ to $1,24{\cdot}10^{-}5\frac{1}{h}$. For the self-healing mode, we modeled the telematics unit to be self-adaption with the success rates $\mu^{n5}_{safe} = 6,85 \cdot 10^{-}4\frac{1}{h}$ and $\mu^{n5}_{sec} = 1,23 \cdot 10^{-}4\frac{1}{h}$ and the firewall to be repairable from defects with the success rate $\mu^{n4}_{safe} = 1,23{\cdot}10^{-}4\frac{1}{h}$. We can see

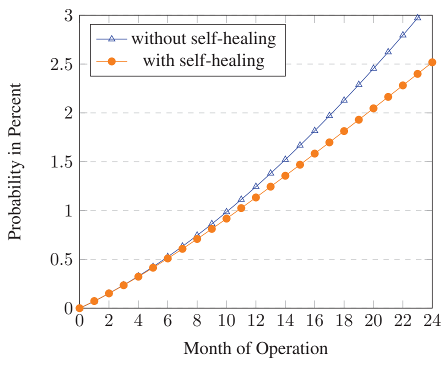

Fig. 5. Failure Probability with/without Self-healing

a noticeable but moderate decrease in the overall failure probability, explained by the fact that self-healing only covers uncritical components that constitute to critical components (e.g. the shown Markov path). Thus, the critical sensor, computing node and actuators can still fail irreparably.

5. Conclusion and Outlook

In the present paper a modeling scheme for self-healing has been established, capturing its main characteristics observable throughout various implementation types, areas and levels. This modeling thereby provides for the properties' generalized consideration in current and future verification methods, which is crucial for its application in critical, real-world systems. Its effectiveness was shown by an implementation in an existing tool and the execution of an example analysis. At the moment, the modeling is limited to the rather simple assumption of randomly distributed failures for indicating the self-healing success. Thus, a refinement of this failure behaviour would be beneficial for achieving a higher and more realistic result precision. While this presents a first step for including self-healing concepts in verification methods, we also discovered that dependencies to other self*-properties like self-configuration and self-protecting exist that have to be considered for an holistic verification. In the future we want to investigate the interplay between the different OC properties and their meaning for system verification.

Acknowledgement

The present work is partially funded by the German BMWE joint project on real driving validation (RDV).

References

Azim, M. T., I. Neamtiu, and L. M. Marvel (2014). Towards self-healing smartphone software via automated patching. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 623–628.

Candea, G. and A. Fox (2001). Recursive restartability: turning the reboot sledgehammer into a scalpel. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pp. 125–130.

Dennis, L. A., R. Monroy, and P. Nogueira (2006). Proof-directed debugging and repair. In *Seventh Symposium on Trends in Functional Programming*, Volume 2006, pp. 131–140.

Ehrig, H., C. Ermel, O. Runge, A. Bucchiarone, and P. Pelliccione (2010). Formal analysis and verification of self-healing systems. In D. S. Rosenblum and G. Taentzer (Eds.), *Fundamental Approaches to Software Engineering*, pp. 139–153. Springer Berlin Heidelberg.

Fischer, P., F. Nafz, H. Seebach, and W. Reif (2011). Ensuring correct self-reconfiguration in safety-critical applications by verified result checking. In *Proceedings of the 2011 Workshop on Organic Computing*, OC'11, NY, USA, pp. 3–12. Association for Computing Machinery.

Güdemann, M., F. Nafz, W. Reif, and H. Seebach (2006). Towards safe and secure organic computing applications. In C. H. et.al. (Ed.), *INFORMATIK 2006 – Informatik für Menschen*, Bonn, Germany, pp. 153–160. K.

Horeis, T. F., J. Heinrich, and F. Plinke (2022). Hybrid modeling for the assessment of complex autonomous systems - a safety and security case study. In M. C. L. et.al. (Ed.), *Proceedings of the 32nd European Safety and Reliability Conference*. Research Publishing.

Kain, T., H. Tompits, J.-S. Müller, P. Mundhenk, M. Wesche, and H. Decke (2020). Fdiro: A general approach for a fail-operational system design. In F. D. M. Piero Baraldi and E. Zio (Eds.), *Proceedings of the 30th European Safety and Reliability Conference and 15th Probabilistic Safety Assessment and Management Conference*. Research Publishing.

Keromytis, A. D. (2007). Characterizing software self-healing systems. In V. Gorodetsky, I. Kotenko, and V. A. Skormin (Eds.), *Computer Network Security*, pp. 22–33. Springer Berlin Heidelberg.

Khalil, K., O. Eldash, A. Kumar, and M. Bayoumi (2019). Self-healing hardware systems: A review. *Microelectronics Journal 93*, 104620.

Monperrus, M. (2018). Automatic software repair. *ACM Computing Surveys 51*(1), 1–24.

Müller-Schloer, C., C. von der Malsburg, and R. P. Würt (2004). Organic computing. *Informatik-Spektrum 27*(4), 332–336.

Nafz, F., H. Seebach, J.-P. Steghöfer, S. Bäumler, and W. Reif (2010). A formal framework for compositional verification of organic computing systems. In B. Xie, J. Branke, S. M. Sadjadi, D. Zhang, and X. Zhou (Eds.), *Autonomic and Trusted Computing*, pp. 17–31. Springer Berlin Heidelberg.

Psaier, H. and S. Dustdar (2011). A survey on self-healing systems: Approaches and systems. *Computing 91*(1), 43–73.

Rinaldo, R., T. F. Horeis, and T. Kain (2021). Hybrid modeling for the assessment of complex autonomous systems - a safety and security case study. In B. C. et.al. (Ed.), *Proceedings of the 31st European Safety and Reliability Conference*, Volume 31. Research Publishing.

Rinaldo, R. and D. Hutter (2020). Integrated analysis of safety and security hazards in automotive systems. In S. K. Katsikas and F. Cuppens (Eds.), *Computer Security*, Volume 12501 of *Lecture Notes in Computer Science*, Guildford, UK. Springer. ESORICS 2020, CyberICPS.

Schleiss, P., C. Drabek, and G. Weiss (2014). Safeadapt deliverable 3.1 - concept for enforcing safe adaptation during runtime. Published at https://www.safeadapt.eu, Accessed 2023-01-26.

Smirnov, A. and T.-c. Chiueh (2005). Dira: Automatic detection, identification and repair of control-hijacking attacks. In *12th Annual Network and Distributed System Security Symposium*.

Stahl, T. (2022). *Safeguarding Complex and Learning-Based Automated Driving Functions via Online Verification*. Ph. D. thesis, Technische Universität München.

Tomforde, S. (2011). *An architectural framework for self-configuration and self-improvement at runtime*. dissertation, Gottfried Wilhelm Leibniz Universität Hannover.

Tomforde, S., J. Hähner, and B. Sick (2014). Interwoven systems. *Informatik-Spektrum 37*(5), 483–487.

Tomforde, S., B. Sick, and C. Müller-Schloer (2017). Organic computing in the spotlight. *CoRR abs/1701.08125*.