Fault Injection Framework for Organic Computing Architecture

Sabikun Nahar, Simon Meckel, and Roman Obermaisser

University of Siegen, Siegen, NRW, Germany

Abstract. In the future, autonomous systems such as self-driving cars must robustly and flexibly handle various fault situations. Static models of faults and countermeasures are standard in classical approaches; however, such static models are no longer efficient as the complexity of fault scenarios tremendously increased. The bio-inspired concept of organic computing applies biological concepts to technical systems. Organic computing utilizes an artificial hormone system as a decentralized mechanism, i.e., a middleware that continuously monitors and organizes task allocations to computing nodes in distributed real-time embedded systems. By introducing different types of artificial hormones for the tasks, task allocations are realized by constantly establishing hormone balances via distributed closed control loops. This process handles the increasing complexity of e.g., distributed control systems by enabling self-configuration, -adaptation, -improvement and -healing. Such an organic computing environment inherently overcomes system-level faults, such as computation node failures, since missing (i.e., non-executed) tasks directly lead to hormone imbalances that are compensated for, thereby restoring these tasks. However, faults in the artificial hormone system, e.g., due to incorrect hormone values, are currently not covered by the organic computing environment and can result in adverse and critical system behavior and even complete system failure. To address these types of faults and thus improve the capabilities and safety of current organic computing systems, this paper presents a fault injection framework to analyze the effects of an extended range of fault cases, including faults in the artificial hormone system. Such analyses are important as they mark the foundation for future fault-handling strategies and safety features in next-generation organic computing systems for autonomous systems. The statistical analyses and results based on the fault injection framework reveal the most safety-related fault cases for which the paper outlines respective fault handling strategies.

Keywords: Fault-tolerant Distributed Computing, Organic Computing, Artificial Hormone System, Hormone Fault Model, Fault Injection Framework

1 Introduction

The high integration density of future embedded systems and their broader range of applications, e.g., in cars and households, are causing them to become increasingly complex. The development, operation, and maintenance of such systems is

therefore a challenge for developers and users. In the context of organic computing, previous research [14] has developed an artificial hormone system (AHS) as a middleware to cope with the complexity of assigning tasks of an application to distributed processing elements (PEs) (also referred to as computation nodes). The AHS uses a hormone-based control loop for task assignments. The respective publications [6] and [7] demonstrate the self-organizing properties of the AHS in terms of stability and real-time capability. In addition, the AHS exhibits reliable behavior against system failures through the use of self-X properties. The concept of organic computing is essentially based on these self-X properties such as self-configuration, self-optimization, and self-healing.

For example, if a processing element crashes during system runtime, the AHS reassigns all tasks of that crashed PE, thus preventing system failure.

However, a faulty processing element may affect the overall task allocations by transmitting (inadvertently) manipulated hormones. Such manipulated hormones, if processed by the hormone loop of each PE, can lead to faulty system behavior, e.g., tasks get allocated too often or are missing.

Since current AHS capabilities cannot overcome such situations, this paper presents a fault injection framework for analyzing and evaluating various fault cases due to hormone manipulations. The fault injection framework enables statistical analyses and worst-case considerations on the impact of faults and is used to design and develop advanced fault recovery strategies that surpass the current organic computing's classical self-X capabilities. The paper is prepared as part of the SelfAutoDOC research project, which aims to implement the principles of organic computing for the first time in a real automotive environment using a steering use case [9].

The paper is structured as follows: After highlighting the motivation of this paper in this section, Section 2 presents the related work. Section 3 describes the organic computing system architecture, the concept of the AHS, and a fault model for the presented system architecture. In Sections 4 and 5, a framework for fault injection is provided and its implementation is then presented in detail. In Section 6, the observations and results of the use cases are analyzed and evaluated. The paper finishes with a conclusion and future work in Section 7.

2 Related Work

The complexity and variability of potential fault scenarios are very high in embedded systems. Systems are needed that automatically and dynamically adapt to an occurrence of such scenarios and initiate appropriate recovery actions to prevent complete system failures. For this, the SelfAutoDOC project proposes the necessary features by applying organic computing, i.e., adapting self-sustaining concepts from biology to technical systems. Systems developed with these concepts have the property of being able to configure, adapt, and heal themselves. In combination with techniques of active fault diagnosis, causes of faults are identified so that, in consequence, the reliability of the system components and the availability of system services can be significantly increased

by means of targeted system interactions, e.g., load distribution and service reconfigurations. The basic principles of self-organizing systems and self-X properties have been addressed by many researchers, e.g., [7], [8], [12].

The DoDOrg project [1] investigated the use of bio-inspired concepts to build a new, self-organizing, robust processor architecture. The artificial hormone system (AHS) was first introduced here. Conferences and workshops such as the ESOS workshop [2] with a focus on organic computing addressed the aspects of self-organization in embedded systems. [15] proposes a formal specification of the hormone loop. As a result, the guarantee of a consistent hormone computation emerged, which is essential for self-Xg properties and task or PE error detection. In 2003, the German Organic Computing Initiative was founded. In brief, it aims to improve the control ability of complex embedded systems based on principles found in organic systems [16]. Another research paper [3] motivates the combination of organic computing based on artificial DNA with adaptive online diagnosis techniques for safety-critical application domains. The approach implements a high-level semantic description for system building blocks, enabling the system to establish itself according to an ADNA building plan (i.e., an application model). Equipped with additional online diagnosis techniques, the system is able to monitor itself and overcome faults and failures by reconfiguration, thus preserving core functionalities.

In contrast to our approach, none of the above approaches addresses fault injection strategies and the development of a fault injection framework for organic computing architectures. The publication [11] demonstrates detection and defense strategies against attacks on an artificial hormone system running on a mixed signal chip. The authors proposed two attacking techniques: one is jamming the AHS with suppressors, and the other is manipulating the quality of task assignment.

3 System Model

3.1 System Architecture

Figure 1 depicts a general architecture of a distributed system, which exhibits self-organizing and self-optimizing properties at runtime. The physical model assumes a distributed system in which the processing elements (PEs) can be heterogeneous. This heterogeneity is accounted for by abstracting from physical components and specifying for each PE which tasks can be performed, specifically, how suitable the PE is to perform a particular task. The logical model is assumed to be a directed acyclic graph, with the vertices of the graph corresponding to the tasks and the directed edges describing the dependencies between tasks.

To evaluate this system architecture, we use a so-called hormone simulator, i.e., a software tool that focuses on task assignments (and neglects inter-task communication). The simulator establishes each PE as an independent thread and implements task assignments through hormone exchange [5]. Local instances

of the artificial hormone system (AHS) run on each PE, i.e., in each thread, and form the middleware for the dynamic assignment of tasks to the heterogeneous PEs. To assign tasks to PEs in a self-organizing manner, the AHS uses three types of hormones to decide on task assignments [4].

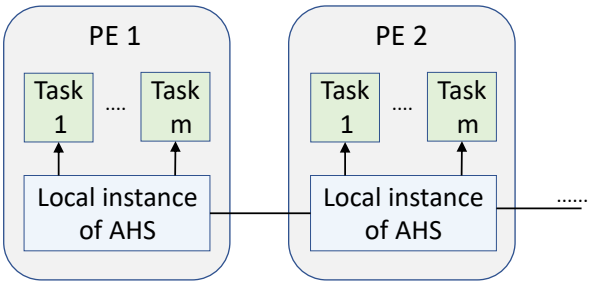

Fig. 1. System architecture

- **Eager hormones** express the suitability of a PE for executing a particular task, thereby abstracting from physical components (such as processor type or memory). For each task of the logical model, a predefined eager value is encoded in the PE, indicating its general suitability. This value is converted into a so-called *modified* eager value, which expresses the momentary suitability, e.g., depending on the current task load of the PE.
- **Suppressor hormones** lower the (momentary) suitability of a PE during the task allocation process. The main purpose of the (global task) suppressors is to indicate a successful task allocation to other PEs in order to prevent duplicate task allocations. There are several types of suppressors, some of which are sent (as messages) to other PEs in the system, while others are used only locally at a PE, e.g., local monitoring suppressors to indicate a deteriorating PE condition, and load suppressors to indicate the current load on a PE from the tasks being executed.
- **Accelerator hormones** can optionally be used to increase the possibility of task execution on a PE. There are different types of accelerators, similar to suppressors, e.g., organ accelerators and monitoring accelerators. The former is the most important used to cluster cooperating tasks among neighboring PEs. The latter is used locally in a PE and indicates improved PE capabilities.

Task assignment is based on hormone flow. Each PE, more specifically the local instance of the AHS on the PE, periodically (and independently) executes a hormone-based control loop [4]. Figure 2 illustrates this hormone loop based on [15], which is a formal specification of the hormone loop of an artificial hormone system. This loop has three stages.

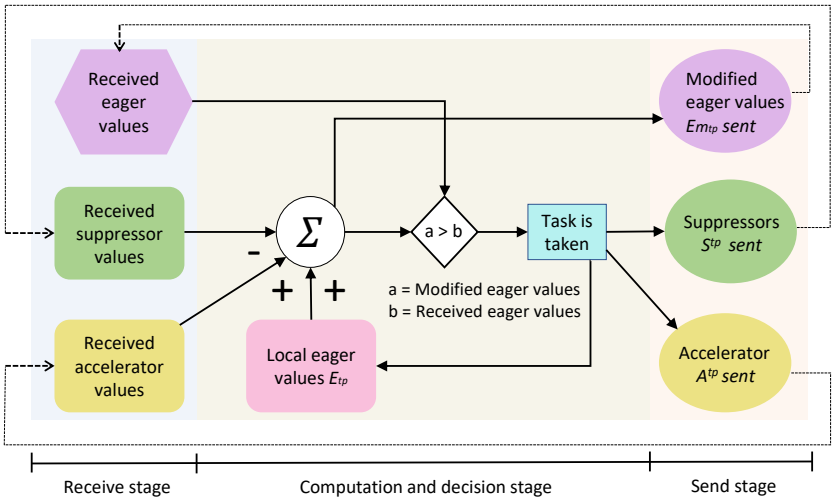

Fig. 2. Hormone-based control loop based on [15]

1. **Receive stage:** In this stage, a PE receives the modified suppressors, accelerators, and eager values, for each task from the other PEs in the system. The former two hormones are used in the equation under point 2) below, whereas the latter hormone is used for a subsequent comparison as described in the following.
2. **Compute and decision stage:** Each PE computes its own modified eager values for each task based on the following equation

$$Em_{t,p} = E_{t,p} - \sum S_{t,p} + \sum A_{t,p}$$

   where t represents task indices and p represents PE indices. $Em_{t,p}$ is the modified eager value of task t on PE p. The local static eager value $E_{t,p}$ indicates how suitable PE p is for executing task t. From this value, all suppressors (resp., accelerators) received for the task are subtracted (resp., added). The resulting modified eager value $Em_{t,p}$ of each task is then broadcast to the other PEs in the send stage. Further, it is compared with the received modified eager values from all other PEs ($a > b$ in Figure 2) to decide on whether to allocate the task. In case of equality, another decision criterion must be used that is known to all PEs, e.g., the unique PE ID.
3. **Send stage:** Once a PE takes over a task t, it distributes suppressors to all PEs dedicated to t, thus preventing other PEs from also executing t, i.e., according to the above equation, a (received) suppressor lowers a PE's suitability. The ratio between an eager value and a suppressor value for a given task determines whether the task is executed once or multiple times in the system. In addition, a PE generates local suppressors to inhibit its

further loading. The PE may also distribute accelerators to other PEs in the neighborhood to attract related tasks and generate local accelerators to prevent task migration.

The described approach is fully decentralized and provides several self-X properties such as self-configuration in terms of initial task allocation by exchanging hormones, self-optimization by re-offering tasks for reassignment when hormone levels change, and self-healing by reassigning tasks to remaining resources in case of PE failures.

3.2 Fault Containment Regions

Fault containment regions (FCRs) define isolated units of failure. They operate correctly and independently regardless of any logical or electrical faults outside the FCR [10]. For the organic computing system architecture shown in Figure 1, we define each PE as an FCR due to using the hormone-based intercommunication, which provides fault isolation for each PE.

3.3 Fault Model

A fault model is a structural representation of faults that may occur in the hardware or software of a system. The fault model in Figure 3 shows the hormone faults this paper addresses in the organic computing architecture. Task assignment is regulated by balancing hormone levels. Faults induce imbalances in hormone levels, which can consequently lead to a relocation of tasks until the balance is reestablished. In this study, we categorize three different hormone faults.

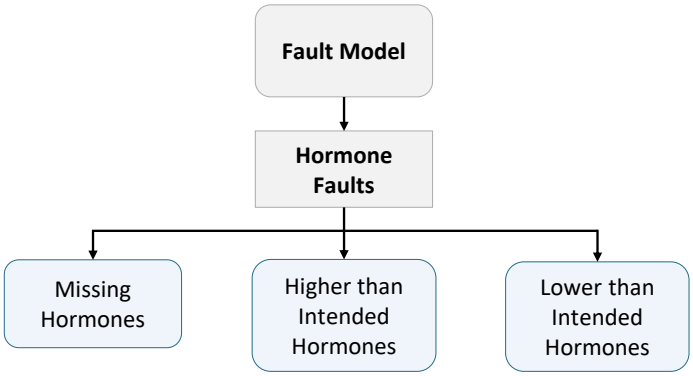

Fig. 3. Flowchart of the fault model

Table 1 exemplifies the variations in the suppressor hormone vector for different fault cases. Injecting a fault means manipulating the hormone values in

the hormone vector. This may affect all tasks, in which case we define the fault as *complete fault*, or only some of the tasks on a PE, in which case we call it *partial fault*.

Table 1. Variations in the suppressor vector

Fault cases	**Original**	**A**	**B**
Missing suppressor	$\begin{bmatrix}40\\40\\40\\40\end{bmatrix}$	$\rightarrow \begin{bmatrix}0\\0\\0\\0\end{bmatrix}$	$\rightarrow \begin{bmatrix}40\\0\\40\\0\end{bmatrix}$
Higher than intended suppressor	$\begin{bmatrix}0\\0\\0\\0\end{bmatrix}$	$\rightarrow \begin{bmatrix}40\\40\\40\\40\end{bmatrix}$	$\rightarrow \begin{bmatrix}0\\40\\40\\0\end{bmatrix}$
Lower than intended suppressor	$\begin{bmatrix}40\\40\\40\\40\end{bmatrix}$	$\rightarrow \begin{bmatrix}3\\3\\3\\3\end{bmatrix}$	$\rightarrow \begin{bmatrix}40\\40\\3\\3\end{bmatrix}$

As an example, suppose that 4 tasks are allocated on a PE. Each task has a predefined suppressor value of 40 represented by the four rows in the **original** vector in the second column. The second vector **A** represents a complete hormone fault injection on a PE (i.e., for all tasks that are allocated to that PE), and the third vector **B** represents a partial fault injection on a PE (i.e., not for all tasks that are allocated to that PE). The following itemization describes all fault cases in detail.

- **Missing suppressors for all tasks:** All non-zero suppressor values of the executed tasks on a PE are set to zero.
- **Missing suppressors for some tasks:** One or more non-zero suppressor values of the executed tasks are set to zero.

- **Higher than intended suppressors for all tasks:** Suppose that no tasks are executed on a PE, resulting in a vector of suppressor values of 0 in the original configuration. As a result of this fault injection on this PE, suppressor values are published for a set of tasks, consequently leading to missing tasks in the system. In this scenario, a higher suppressor value than intended is published for all tasks.
- **Higher than intended suppressors for some tasks:** For the tasks that are not executed on a PE, one or more suppressor values are published.
- **Lower than intended suppressors for all tasks:** All suppressor values of the executed tasks are set to lower values than intended.
- **Lower than intended suppressors for some tasks:** One or more suppressor values of executed tasks are set to lower values than intended.

4 Fault Injection Framework

Fault injection refers to an intentional introduction of faults or errors into a target system through controlled experimentation [13]. As a result of fault injection, the user can observe and analyze the effects of faults on the target system. In this study, we use simulation-based fault injection.

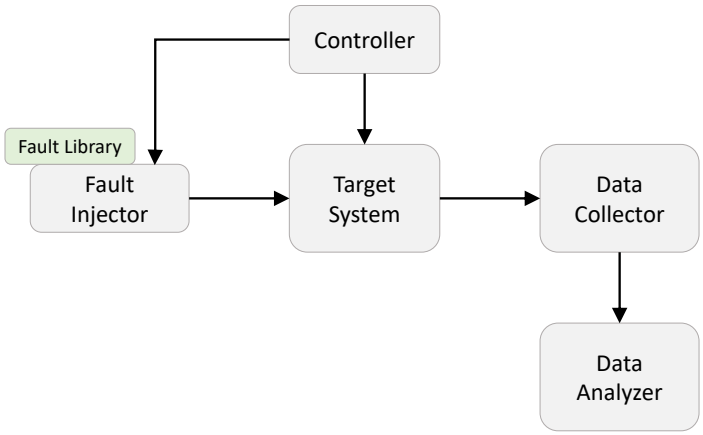

Fig. 4. Basic components of the fault injection framework

The fault injection framework is a term used to describe the various components that perform a fault injection campaign. These components include *fault injector, fault library, controller, data collector* and *data analyzer.* The *fault injector* is the component that introduces faults into a target system. In our case, the *fault injector* is determined by a fault model and contains a *fault library.* The *fault library* stores the set of fault cases, fault locations, and fault times for

the fault injection. The *controller* contains a workload library that stores sample workloads for the target system. It triggers the fault injection by switching on the fault cases and automatically changing the fault locations in each new experiment. The *data collector* collects the data observed during experiments and stores it in different files. Then, the *data analyzer* examines the data and retrieves only the essential information for analysis. Figure 4 shows the model of the fault injection framework illustrating the basic components and the relationship between them.

5 Implementation

In general, the AHS simulator is implemented to demonstrate and evaluate the behavior of the AHS approach. This section focuses on the fault injection component, which is integrated into the AHS simulator. When running on a general-purpose computer, the simulator creates independent threads for each processing element and assigns tasks by exchanging hormones. There are several modules in the AHS simulator. Every module is coded in ANSI C in a platform-independent manner [17]. The fault injection component, known as the fault injector, contains three types of faults.

Algorithm 1 Missing hormone fault model algorithm

```
function MissingHormoneFailureModel()
    Switch (case number)
    while case 0: do
        No fault is triggered. Do the basic sending.
        break;
    end while
    while case 1: do
        if Current timestamp >= Trigger timestamp then
            if Processing Element is faulty = True then
                Get the active tasks of faulty PE and send
                suppressor value 0 on all PEs dedicated to
                the same task.
            end if
        end if
        break;
    end while
end function
```

1. Missing hormone fault: The missing hormone fault model is a function coded in the main file of the hormone simulator. This function is called during the send stage of the hormone loop, specifically while the suppressor value is being sent to all PEs. As a result, the active task of the targeted PE sends

a suppressor value of 0 to all PEs dedicated to the same task. A suppressor value 0 specifies that there is no suppressor value available at all. Algorithm 1 summarizes the operation of the fault logic.

2. Higher than intended hormone fault: The higher than intended hormone fault is implemented by a function that is triggered during the calculation of the suppressors to be sent for each task. This fault is also coded in the main file of the hormone simulator. Due to this fault, a suppressor value is sent by the faulty PE for all the tasks that are not yet assigned to any PE. As a consequence, these tasks are missing in the system. Algorithm 2 shows the implementation of the fault logic.

Algorithm 2 Higher than intended hormone fault model algorithm

```
function higherThanIntendedHormoneFaultModel()
    Switch (case number)
    while case 2: do
        if Current timestamp >= Trigger timestamp then
            if Processing Element is faulty = True then
                Send a higher suppressor value from the
                faulty PE to other PEs for all the tasks.
            end if
        end if
        break;
    end while
end function
```

3. Lower than intended hormone fault: The lower than intended hormone fault is implemented by a function that sends a low suppressor value during the send stage of the hormone loop. The implementation of this fault is similar to the missing hormone fault. Missing hormone defines a value of 0, which means that there is no hormone available while the lower than intended hormone fault defines a value between 0 and the correct hormone value.

6 Experimental Results

This section discusses the observed behavior for the three types of fault cases (i.e., missing suppressor hormone fault, higher than intended suppressor hormone fault, and lower than intended suppressor hormone fault) in the organic computing simulation environment. We analyzed the task allocation of the system after each fault injection. The following graphs highlight the results of the experimental evaluations.

The goal of the experiments performed here is to test the simulation of the organic computing architecture, including the control of the fault injection timing and its ability to target the task allocation.

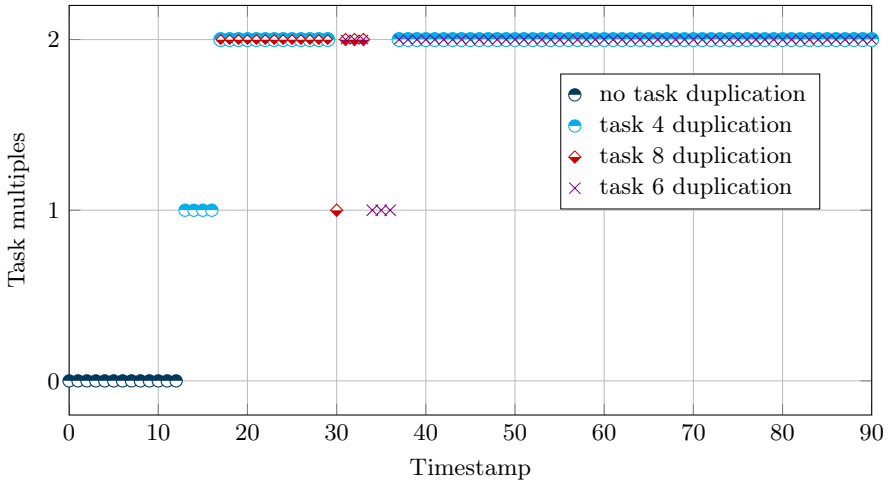

Fig. 5. Multiple tasks due to missing suppressor hormone fault injection

Figure 5 graphically depicts the duplicate task allocation due to the missing suppressor hormone fault injection in the system. Each different color and shape refers to a specific task id. The system configuration exemplarily comprises 4 processing elements and 8 tasks. The fault is injected in the 3rd PE at timestamp 10. During initial task allocation, all tasks are mapped to PEs in such a way that each PE executes two tasks. Table 2 shows the event that occurred in Figure 5.

Table 2. Event description occurred in Figure 5

Timestamp	**Event occurred**
1–9	8 tasks are allocated on 4 PEs. Each PE executes 2 tasks.
10	Fault triggered.
13–16	Task 4 from faulty PE is duplicated.
17–29	Task 8 is offered and taken by faulty PE. Two duplicate tasks (i.e., of task ids 4 and 8) are running.
30	Duplicated task 4 from faulty PE is loose pending and lost. Only the duplicate of task 8 remains.
31–33	Task 6 is offered, taken by faulty PE and duplicated. Two duplicate tasks (i.e., of task ids 6 and 8) are running.
34–36	Duplicated task 8 from faulty PE is loose pending and lost. Only one duplicated task 6 remains.
37–100	An offered task 4 is taken by faulty PE and duplicated whereas duplicate task 6 also remains.

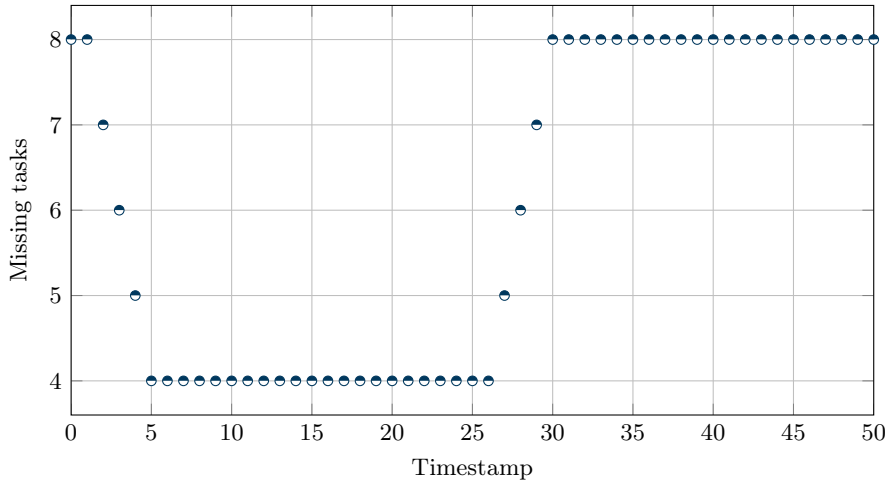

Fig. 6. Missing tasks due to higher than intended suppressor hormone fault injection

The result in Figure 6 for higher-than-intended suppressor hormone fault injection explores missing tasks in the system. The system configuration is similar to used for Figure 5 except for the suppressor hormone value and the fault injection time. At timestamp 5 the fault has occurred in the 3rd PE. A higher suppressor value is sent for the tasks before those tasks are initially allocated to the PE. Table 3 shows the event description that occurred in Figure 6.

Table 3. Event description occurred in Figure 6

Timestamp	**Event occurred**
1–4	From a total number of 8 tasks, 4 tasks are running on 4 PEs.
5	Fault triggered. Remaining 4 tasks have gone missing out of 8 tasks.
27–29	Each running task goes missing from the system once offered.
30–50	All 8 tasks are missing from the system.

Figure 7 graphically shows the duplicated task allocation due to the lower-than-intended suppressor hormone fault injection in the system. As the legend indicates, each task id is highlighted in different colors and shapes. Based on the fault type Figure 7 also demonstrates similar impacts as shown in Figure 5. During simulation, the system is configured with 4 PEs and 8 tasks. At timestamp 10 the fault occurred in the first PE. A lower suppressor value of 3 is sent from the faulty PE to all other PEs dedicated to the same task. A detailed description of the events that occurred in Figure 7 is shown below in Table 4.

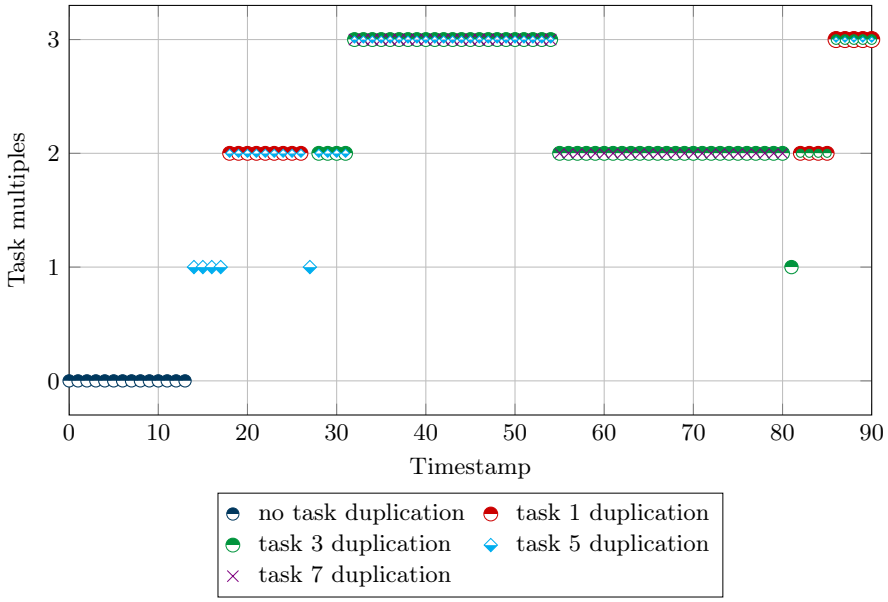

Fig. 7. Multiple tasks due to lower than intended suppressor hormone fault injection

Table 4. Event description occurred in Figure 7

Timestamp	Event occurred
1–9	8 tasks are assigned to 4 PEs and each PE has taken 2 tasks.
10	Fault triggered.
14–17	Task 5 from faulty PE is duplicated.
18–26	Tasks 1 and 5 from faulty PE are duplicated.
27	Duplicated task 1 from faulty PE is set to loose pending and lost. Only one duplicate (of task 5) remains in the system.
28–31	There are offered tasks in the system which is taken by faulty PE and gets duplicated. Task 3 is taken by faulty PE and duplicated.
32–54	The offered task 7 is taken by the faulty PE and duplicated. As a result, the system is running duplicates of the tasks 3, 5, 7.
55–80	Duplicated task 5 from faulty PE is set to loose pending and lost. The other two duplicated tasks 3 and 7 remain in the system.
81	Duplicated task 7 from faulty PE is set to loose pending and lost. Only one duplicate (task 3) remains.
82–85	The offered task 1 is taken by the faulty PE and duplicated. Again, two duplicates (of tasks 1 and 3) are running.
86–100	Task 5 is offered again, taken by faulty PE and duplicated. As a result, three duplicates (of tasks 1, 3, 5) are running.

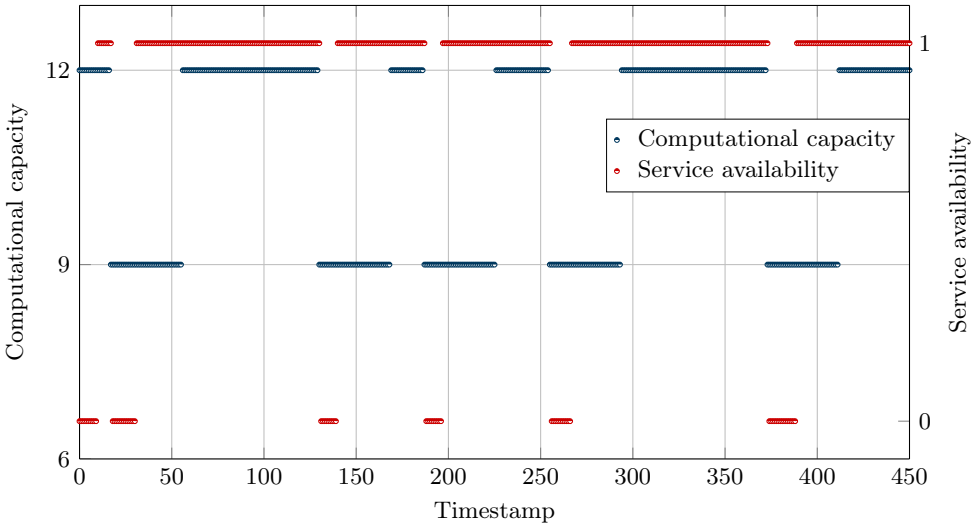

Fig. 8. Service availability with PE crash

As a further experiment, a simulation was conducted to evaluate the service availability in an environment where PEs fail according to statistical failure rates (and come back online after a certain outage time). The setup is based on 4 (homogeneous) PEs, each able to host up to 3 tasks. The *computational capacity* is defined as the number of tasks that can be hosted, so overall, the distributed system has a computational capacity of 12 (tasks). The crash probabilities of the four PEs at every timestamp are given as percentages of 0.1 %, 0.2 %, 0.3 %, and 0.4 %, respectively. According to the working principles of the AHS, this leads to task re-allocations, every time a PE crashes. The measure of service availability for the system can be simplified as follows (assuming broadcast-based communication among tasks): Let there be 9 tasks. If all of them are allocated, the system service is available (i.e., the red points corresponding to the left horizontal axis in Figure 8 are one) and it is not available if at least one task is missing (i.e., the red points corresponding to the left horizontal axis in Figure 8 are zero).

In the figure, one sees that as soon as one PE crashes (i.e., the computational capacity represented by the blue point drops from 12 to 9), the system service becomes unavailable for a certain maximum number of hormone cycles until the AHS reallocates the missing tasks to the remaining PEs. This is possible since the remaining computational capacity of 9 is sufficient, resulting in a reestablished system service.

In order to demonstrate the limitations of the current organic computing's AHS middleware, the above experiment was extended in a way that not only PEs fail according to statistical failure rates, but also the fault type *missing suppres-*

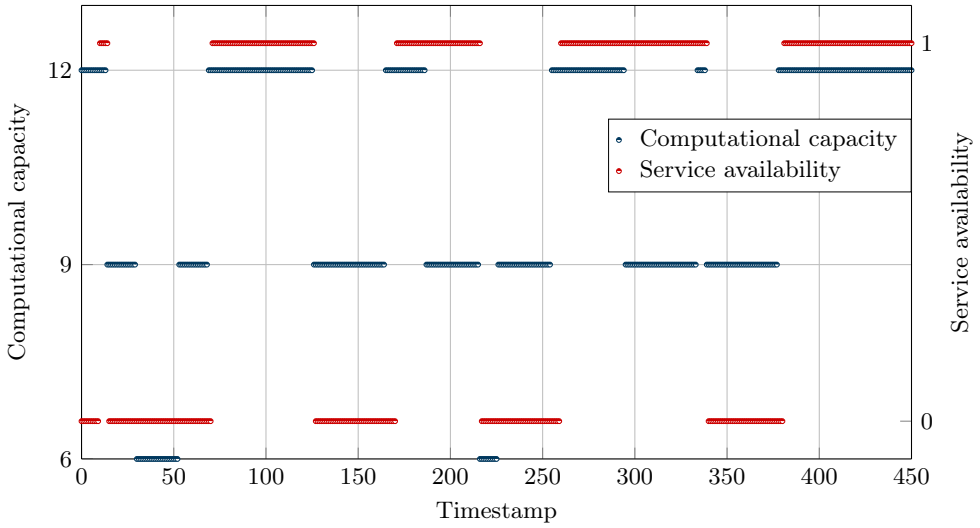

Fig. 9. Service availability with PE crash and missing suppressor hormone fault injection

sor hormone fault (recall Section 3) is additionally injected at timestamp 10 in Figure 9. As a consequence, all tasks currently assigned to the faulty PE (i.e., the PE where the local instance of the AHS does not transmit suppressors anymore), will be duplicated. These duplicated tasks consume computational capacity on other PEs which means that, in contrast to the result of Figure 8, the system service might not be available even if the computational capacity is at least 9. On the other hand, as per the above definition of system service availability (i.e., all 9 tasks must be executed), one can have cases where the service availability remains on one (red line in Figure 9), even if the computational capacity drops from 12 to 9 (e.g., at timestamp 300).

Overall, comparing Figures 8 and 9, it can be concluded that classical organic computing's AHS extension is necessary to improve self-healing capabilities to overcome hormone fault cases.

7 Conclusion and Future Work

This paper, prepared as part of the ongoing SelfAutoDOC research project, addresses the research gap of developing next-generation organic computing systems that enable fail-safe applications for autonomous driving requirements. For this, the paper implements a fault injection framework to test and evaluate an extended range of faults, beyond what is currently handled in classical organic computing. First, a fault model that focuses on faults of artificial hormones (i.e., in the organic computing middleware) is presented. A fault injection framework

was then designed and implemented. It enables statistical analysis of the effects of hormone faults in the organic computing middleware and reveals the most safety-relevant hormone fault cases for bringing organic computing into the domain of autonomous systems. The paper outlines corresponding fault handling strategies and improvements to the self-X properties that will help move the organic computing system towards the required high reliability and availability in future autonomous driving systems. The next research and development activities in SelfAutoDOC include the design and implementation of a so-called hormone guard that is able to contain hormone faults in the time and value domain.

Acknowledgment

This work was supported by the research project SelfAutoDOC funded by the Bundesministerium für Wirtschaft und Klimaschutz (BMWK) under grant number 19A21044C. The hormone simulator used for this work was developed by the Chair for *Eingebettete Systeme* at Goethe University of Frankfurt am Main, Germany.

Gefördert durch:

aufgrund eines Beschlusses
des Deutschen Bundestages

References

1. Jürgen Becker, Kurt Brändle, Uwe Brinkschulte, Jörg Henkel, Wolfgang Karl, Thorsten Köster, Michael Wenz, and Heinz Wörn. Digital on-demand computing organism for real-time systems. In *ARCS Workshops*, volume 81, pages 230–245, 2006.
2. Uwe Brinkschulte, Christian Müller-Schloer, and Mathias Pacher. Workshop on embedded self-organizing systems, san jose, usa, 2013.

3. Uwe Brinkschulte, Roman Obermaisser, Simon Meckel, and Mathias Pacher. Online-diagnosis with organic computing based on artificial dna. In *2019 First International Conference on Societal Automation (SA)*, pages 1–4. IEEE, 2019.
4. Uwe Brinkschulte and Mathias Pacher. An agressive strategy for an artificial hormone system to minimize the task allocation time. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. IEEE, 2012.
5. Uwe Brinkschulte, Mathias Pacher, and Benjamin Betting. A simulator to validate the concept of artificial dna for self-building embedded systems. In *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 160–169. IEEE, 2014.
6. Uwe Brinkschulte, Mathias Pacher, and Alexander von Renteln. Towards an artificial hormone system for self-organizing real-time task allocation. In *IFIP International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, pages 339–347. Springer, 2007.
7. Uwe Brinkschulte, Mathias Pacher, and Alexander von Renteln. An artificial hormone system for self-organizing real-time task allocation in organic middleware. In *Organic Computing*, pages 261–283. Springer, 2009.
8. Carlos Gershenson. *Design and control of self-organizing systems*. CopIt Arxives, 2007.
9. Timo Kisselbach, Simon Meckel, Mathias Pacher, Roman Obermaisser, and Uwe Brinkschulte. Organic computing to improve the dependability of an automotive environment. In *Proceedings of the International Conference on Architecture of Computing Systems (ARCS)*, 2022.
10. Jaynarayan H. Lala and Richard E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82(1):25–40, 1994.
11. Christoph Leineweber, Mathias Pacher, Benjamin Betting, Julius von Rosen, Uwe Brinkschulte, and Lars Hedrich. Detection and defense strategies against attacks on an artificial hormone system running on a mixed signal chip. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 135–143. IEEE, 2012.
12. Gero Mühl, Matthias Werner, Michael A. Jaeger, Klaus Herrmann, and Helge Parzyjegla. On the definitions of self-managing and self-organizing systems. In *Communication in Distributed Systems – 15. ITG/GI Symposium*, pages 1–11. VDE, 2007.
13. Thomas Naughton, Wesley Bland, Geoffroy Vallee, Christian Engelmann, and Stephen L Scott. Fault injection framework for system resilience evaluation: fake faults for finding future failures. In *Proceedings of the 2009 Workshop on Resiliency in High Performance*, 2009.
14. Mathias Pacher and Uwe Brinkschulte. Implementation and evaluation of a self-organizing artificial hormone system to assign time-dependent tasks. *Concurrency and Computation: Practice and Experience*, 24(16):1879–1902, 2012.
15. Mathias Pacher and Uwe Brinkschulte. A formal specification of the hormone loop of an artificial hormone system. In *ESOS*, 2013.
16. Hartmut Schmeck. Organic computing – a new vision for distributed embedded systems. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. IEEE, 2005.
17. Alexander von Renteln and Uwe Brinkschulte. Implementing and evaluating the ahs organic middleware – a first approach. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 163–169. IEEE, 2010.