Time-Triggered Organic Computing Architecture for Autonomous Driving Vehicles Using List Scheduling

Mario Qosja, Simon Meckel, Roman Obermaisser
Chair for Embedded Systems, University of Siegen, Siegen, Germany
mario.qosja@uni-siegen.de, simon.meckel@uni-siegen.de, roman.obermaisser@uni-siegen.de

Abstract

As the autonomous vehicles market is expected to grow in the future, their functionalities will increase too, leading to complex embedded computer systems. To address this, organic computing has emerged as a research area that takes inspiration from biological entities to handle complex distributed embedded computer systems. Organic computing has improved adaptability and robustness, while also reducing development efforts. However, it has some drawbacks in terms of determinism, composability, and dependability, which are key features for safety-critical applications. Distributed computer systems using time-triggered communication networks possess these characteristics and thus show distinct advantages for safety-critical systems. By combining artificial DNA and hormone features with time-triggered communications, we can make these systems safer, more reliable, and suitable for safety-critical applications. Therefore, we present a time-triggered organic computing architecture where the time-triggered schedule is produced by a list scheduling algorithm during runtime. All the tasks on the systems are executed according to the computed schedule. To demonstrate the concepts and the system model, we performed evaluation test in a simulator in which the system executes tasks according to predefined schedules. The evaluation of use cases shows improved temporal predictability and fault containment.

1 Introduction

The market demand for autonomous vehicles has increased in recent years and is expected to grow exponentially in the future. This increase in demand will result in vehicles with complex functionalities, leading to challenges in their handling, especially in fault situations. In current designs, the human driver provides adaptability and flexibility in challenging driving situations, so driver assistance functions do not need to be fail-operational. As we move towards self-driving vehicles, more and more driver assistance functions will need to be fail-operational, i.e., they will need to provide system services even in the event of faults (such as failure of computing nodes or sensors). Therefore, we need computing systems that can dynamically and autonomously adapt to these complex situations and provide sufficient redundancy at a limited cost. IBM defines systems as autonomous if they contain self-x properties (such as self-organization, self-healing, self-configuration) [9]. Organic Computing (OC) is a paradigm for organizing distributed computer systems with a high degree of flexibility and self-healing, which is inspired by the concepts and principles of biological systems. It adapts the working principles of organic systems to manifest their self-organizational nature into complex embedded systems. This is done by employing an Artificial Hormone System (AHS) as a real-time middleware that brings improved adaptability and robustness by exhibiting a self-organizing mechanism that can self-configure and heal the system. This is achieved by exchanging artificial hormones (i.e., small messages) between all computing nodes in a distributed system to determine the suitability of task allocations, initially in the start-up phase (self-configuration), in the event of node failures (self-healing) and after potentially degrading system services (reconfigurations). However, the AHS currently lacks support for dependability, determinism, and composability, which are crucial for safety-critical systems such as autonomous driving systems. On the other hand, this is provided by time-triggered systems. Such systems ensure resource adequacy and predictability through a priori scheduled tasks and messages. Knowledge of the permitted temporal behavior of components allows effective fault containment in the time domain and simplifies certifiability. In addition, in this paper a Time-Triggered Organic Computing (TTOC) architecture for the automotive domain has been presented that combines the advantages of both organic computing and time-triggered systems. The flexibility in terms of task (re-)allocations is maintained by the artificial hormone system and the predictability is realized by a list scheduler that will organize task execution times, message injection times, and message paths. Furthermore, it deals with the distributed scheduling problem in the TTOC environment for each computing node. In typical designs, the scheduler organizes the temporal and spatial allocation of both tasks and messages, but in the proposed architecture the AHS middleware handles the allocation of application tasks to ECUs meanwhile the scheduling algorithm will handle execution times and message paths. The paper is divided as follows: In section 3 it provides a short introduction to the specifics of OC and gives an overview of the combination of time-triggered concepts with OC. Section 4 will discuss the scheduling problem for the TTOC and the newly proposed algorithm designed by us. Section 5 describes the evaluation of concepts that have been proposed using the TTOC simulator. Section 6 draws the conclusion and the future work.

2 Related Work

Self-X properties of autonomous systems have been an area of research that has received significant attention over the past few years. Organic Computing was established as a

research field by Deutsche Forschungsgemeinschaft (German National Science Foundation) in 2003 [8] to bring the principles of biological systems into distributed computer systems. A distributed self-organization OC based on the observer/controller architecture is introduced in [12] with the ability to control unexpected behaviors of the system. Another observer/controller design is developed in [3]. This OC system is tested on a traffic light controller. In addition, there have been OC constructed from different approaches. Similar to how genetic instructions encode the functioning and growth of organisms, ADNA can also encode the structure and organization of embedded computer systems [4] and store it inside each computing node (ECU), following the same principles as biological DNA. An additional, organic computing technology inspired by biological systems has been established at [7]. It is a real-time middleware based on an artificial hormone system. This middleware exhibits self-organizing property, allocating tasks to the most suitable processing node by itself. AHS uses artificial DNA to construct distributed embedded systems.

Furthermore, significant research has been conducted to introduce self-organization and OC properties in the automotive domain. OC will handle the increasing complexity of embedded systems in the autonomous vehicle sector. The dynamic concepts of ADNA and AHS can be used in AUTOSTAR [6]. Moreover, organic computing has also been used to improve the dependability of automotives as described in [10]. About the middleware layer, an autonomic middleware for automotive embedded systems that exhibits high flexibility and automatic runtime reconfigurations is presented in [1]. An alternative middleware approach [2] dynamically configures automotive embedded systems by providing transparency and flexible platform-independent support for portability.

The proposed architecture combines time-triggered concepts with organic computing, based on artificial hormone system middleware for autonomous vehicles. This time-triggered OC middleware achieves high flexibility and reliability that comes from the self-x properties and predefined task execution. The schedules get dynamically calculated for each computing node by a heuristic scheduling algorithm.

3 Time-Triggered Organic Computing Architecture

We have taken inspiration from the concept of ADNA and AHS, two organic computing technologies following the same philosophy as biological systems. ADNA and AHS offer several advantages, such as robustness, reduced development efforts, and increased adaptability. In section 1, we proposed combining ADNA and AHS with time-triggered concepts to make embedded systems more deterministic and reliable. We call this new architecture Time-Triggered Organic Computing (TTOC). To begin with, in this section, we provide a brief overview of ADNA and AHS, followed by a detailed explanation of the new architectural concepts.

3.1 Artificial DNA

Different complex embedded systems can be constructed by inserting the structure and organization of the system in a single file and storing it in each computing node [5]. The functionalities (e.g., task structures and messages) are encoded in the ADNA file using simple basic elements.

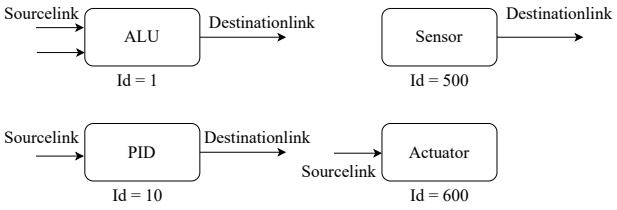

Figure 1 Different basic elements

Figure 1 exemplarily shows functional basic elements of an embedded system (e.g., filters, actuators, sensors) where *Sourcelink* denotes a reactive link that responds to incoming requests and *Destinationlink* is an active link for sending requests. In the automotive domain, a basic element can also be, for example, the ABS functionality. In the current designs, the ABS is bound to a fixed ECU and a backup one. In case of a failure in both ECUs, the system loses the ABS functionality, which may lead to wheels locking up during breaking. With the ADNA file located in each processor core, the ABS functionality will be transferred to other active ECUs, thus preventing the lockup of the wheels.

3.2 Artificial Hormone System

The AHS middleware is designed to read the ADNA computer file and create system functionalities. It can also implement other self-x features, such as self-configuring, where the system reconfigures itself during run-time. Once the tasks are created, AHS will allocate them to specific computing nodes based on their suitability, organizing the system (self-organization) accordingly. For example, in control loops, tasks from the PID controller will have higher suitability on computing nodes that perform arithmetic calculations better.

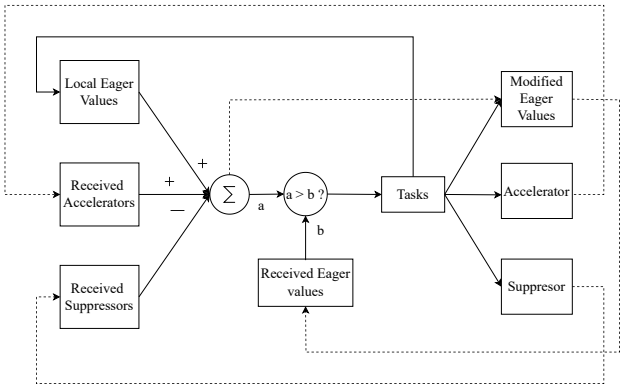

Figure 2 Hormone Loop of AHS

Furthermore, AHS also distributes tasks to minimize communication distances (self-optimizing). Each ECU ex-

ecutes an hormone loop for sending hormones and calculating task suitability from these hormones. Each task contains three main hormone types: *Suppressors* for lowering suitability, *Accelerators* for increasing suitability, and *Eager values* for determining task suitability of the ECU. The hormone loop shown in **Figure 2** sums up all received accelerators, suppressors, and local eager values, and compares these results with received eager values. AHS advantages rely on self-x properties such as self-organization, self-building, self-optimization, and self-healing. In AHS, the system is self-organized because task distribution is done internally by exchanging hormones between the nodes, taking into consideration the suitability of tasks but also the load of a node. The system becomes more optimized by preventing electronic control units from experiencing high loads. The entire system is built using simple artificial DNA files, which allows for reconfiguration in the event of computing node failure. In such cases, all tasks are migrated to other computing nodes.

3.3 Time-Triggered Architecture

Distributed embedded computer systems are built using nodes consisting of communication controllers and computer hosts that communicate via time-triggered networks. In the TTOC, each ECU will serve as one node of the system as presented in **Figure 3**.

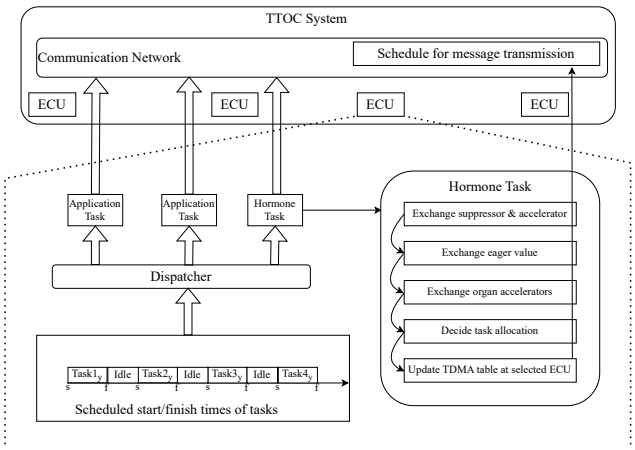

Figure 3 Time-Triggered Organic Computing Architecture

In organic computing based on ADNA, different ECUs concurrently perform various tasks, such as sensor or actuator tasks. In distributed embedded systems, ensuring a guaranteed consistent system behavior is crucial. This is achieved by processing events on all nodes in the same consistent order. Using a global time base to execute operations ensures the same order for all nodes, which brings determinism to the system. Being predictable makes the systems deterministic, thereby increasing overall reliability. The global time is also utilized for error detection, communication protocols, and interfaces of the nodes. For instance, in the case of a babbling idiot failure, where a computing node sends untimely messages, all nodes have predefined times when they can communicate. If communication occurs outside the specified time, bus guardians will block the messages that occur outside of the predefined time slots. In TTOC, any communication network that supports time-triggered concepts is suitable. At the operating system layer, each ECU has a task dispatcher and a time-triggered schedule. The dispatcher is responsible for reading the schedule and executing different tasks at particular points in time. A schedule table with the start and finish execution times of the tasks is shown in Figure 3. In the architecture, alongside the application tasks from the application model, such as sensor data, there are also the TTOC middleware tasks composed of hormone message exchange, which is also time-triggered. The ECU can also be in an idle stage, where it waits for the next task to be processed.

4 Scheduling

There are two types of real-time applications, hard and soft. If the failure of meeting the deadline causes a fatal fault this is called a hard deadline. It is called soft where missing the deadline will not have a big impact on the application. For autonomous vehicle systems, if the deadlines are missed, the caused failure can lead to vehicle malfunction. To be able to ensure that all application tasks will meet their deadlines, real-time scheduling must be performed. The scheduling algorithm determines the order of the tasks that are going to be processed by the computing system.

There are two types of scheduling algorithms: static and dynamic. If the task priorities and the execution times are determined before the program starts, the scheduling is static. In dynamic scheduling, everything must be calculated during the run-time of the system. The scheduler comes up with a scheduling algorithm for the system, which is in two forms: preemptive or non-preemptive. In the preemptive form, if a task with a lower priority has blocked one with a higher one, the process with lower priority will be terminated, allowing the execution of the higher one. In the non-preemptive the process does not terminate but it waits until the CPU burst time is complete. In classical designs, the scheduler organizes:

- Allocation of application tasks to computing nodes
- Application task execution times
- Message injection times
- Messages paths

In the proposed architecture, this process is split and performed incrementally. The AHS middleware first determines the allocation of tasks (of the application) in the system at run-time, without execution times, thus allowing for the desired flexibility that saves hardware cost, yet realizing the same level of redundancy. The list scheduling algorithm in TTOC then dynamically calculates the tasks execution times along with the message injection times and paths for the application messages, according to the given tasks assignments. The hormone exchange middleware tasks that run on each ECU are also time-triggered, but their timing within the schedule period is predefined.

4.1 List Scheduling

As mentioned, in TTOC the scheduling algorithm must be computed at each ECU dynamically without knowing the priorities of tasks. The dynamic scheduling algorithm that has been taken into consideration for TTOC is List Scheduling (LS). List scheduling is a process used to schedule tasks represented as a directed acyclic graph (DAG).

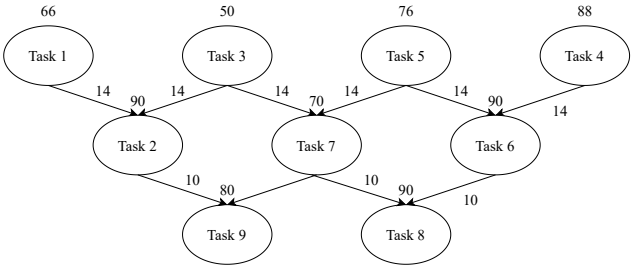

Figure 4 Example of DAG graph

There is a DAG graph consisting of nine application tasks and their worst case execution times (wcet) alongside message size as shown in **Figure 4** . The tasks are arranged based on their depth in the graph and the time required for their execution. Each task is represented by a node and dependencies between them are represented by edges in the graph. The scheduling algorithm determines the order in which the tasks should be executed and aims to assign each task to an available computing unit (ECU). It takes into consideration the data dependencies, timing constraints, system model, critical path, scheduling priorities, and optimizing for factors such as load balancing together with minimizing idle time. Since the spatial allocation is taken care of by TTOC middleware, the AHS will start processing the hormone loop tasks based on their execution order. For that reason, it is important to analyze and set the execution order of tasks.

In our above DAG graph (Figure 4), task two is data dependent from task one and task three because the messages that arrive from them are needed for its computations. In this case, task two must wait for the messages coming from task one and task three before it can start executing. Task one and task three are denoted as immediate parents of task two. For the timing constraints, as it is written in the entry of the section, in our case our system must meet hard deadlines. For the system model, Figure 3 describes that each ECU will execute the scheduling algorithm and produce its own schedule. In addition, the critical path in the DAG is calculated by taking into consideration the computation time (WCET) of ECU and the communication times (message size) on that particular path. Hence, we start processing tasks from the most critical path of the DAG. The b-level has been used to determine the node levels and scheduling priority.

After the tasks are ordered, the AHS can start the hormone loop to determine the suitability levels of the task and assign it to the most suitable ECU. Only then the scheduling algorithm can produce the temporal allocation of the application tasks and the (spatial + temporal) allocation of messages. The algorithm below describes the procedure for schedule generation of each task. As an input, the algorithm will take the application task that has to be scheduled. As stated, the TTOC middleware tasks will also be executed in a time-triggered manner, which is why they will have predefined slots on the schedule in each period. The period of these fixed slots is denoted as the TTOC period and the whole period as the hyper period. This means that all the application tasks and messages will need to be scheduled after the TTOC period, otherwise the AHS will not allocate application tasks to ECU and the whole system will malfunction. For this reason, in the algorithm, it is constantly checked if the start time, or the start time plus the width of the application tasks, does not overlap with the TTOC period.

```
Algorithm Schedule(ECU)
    Data: Task to be scheduled
    for all tasks parents do
        for Parent sending messages do
            if Message receivers Id == Task ID
             then
                if Tasks run on same ECU then
                    Message schedule generation;
                    Update TDMA table of ECU ;
                end
                if Tasks run on different ECU then
                    Message schedule generation;
                    Update TDMA table of ECU;
                end
            end
        end
    end
    if Task is not scheduled then
        Task start time == ECU time ;
        if Task is the first one to be scheduled then
            Task schedule generation;
            Update TDMA table of ECU;
        end
        if Task start time < hyper period + TTOC
         period then
            start time = hyper period + TTOC
             period;
            Task schedule generation;
            Update TDMA table of ECU;
        end
        if Task start time + task wcet > hyper
         period + TTOC period then
            start time = 2 * hyperperiod +
             TTOCperiod ;
            Task schedule generation;
            Update TDMA table of ECU;
        end
    end
```

Furthermore, the complexity increases since the schedule is calculated for one task at a time. Since the allocation of the tasks that receive messages from the task that is being scheduled is not present till the AHS assigns it on an ECU, only the temporal allocation can be calculated.

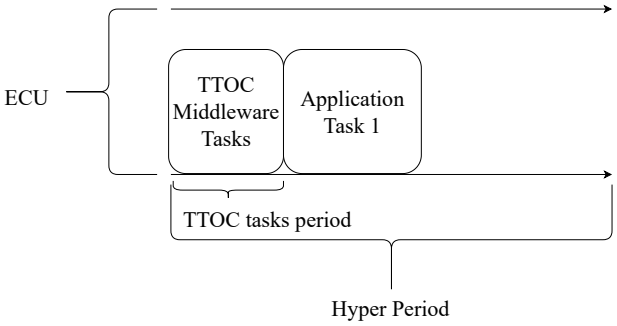

Figure 5 Schedule generation example per each ECU

The solution proposed is to first schedule all the incoming messages from all the task parents and then continue with the application task schedule. **Figure 5** represents the schedule generation phase for each ECU that happens dynamically while the system is running.

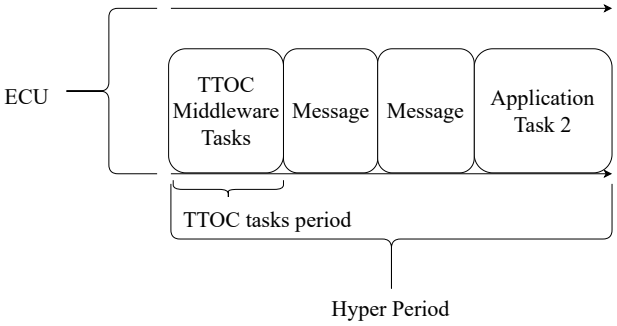

Figure 6 Schedule generation example per each ECU

5 Evaluation of use cases

In the evaluation part, we tested the new scheduling algorithm, the time-triggered concepts of TTOC, and the self-x properties of OC. The simulations were conducted on the TTOC simulator developed by [11]. In this system, the ECUs are implemented as processing elements that contain a task dispatcher, which utilizes a schedule to execute AHS middleware tasks or dummy application tasks.

5.1 Testing List Scheduling

For the testing of a list scheduling algorithm, the TTOC simulator has been used. In this simulator, automotive application tasks are represented in the form of dummy tasks. For the schedule calculation, multiple JSON files, that contained different applications and physical models, were used. The range of application models varies to examples up to forty tasks and with a total of fifty messages. All the results, from the ordering of tasks, spatial and temporal allocations, and execution times of the dispatcher are dumped into a log file. The analysis of the log file concludes that the schedule is generated dynamically as intended on each ECU.

5.2 Testing Time-Triggered Concepts

For the time-triggered concepts, we are interested in the system executing application tasks and communication of messages, according to the generated table schedules. As stated in this section, in the TTOC simulator each ECU contains a task dispatcher that operates based on the time-triggered schedule. To evaluate the reliability of the TTOC architecture, babbling idiot failure was tested on the system. A random ECU was chosen to send untimely communication messages to the other ECUs. The results showed that the failed ECU was blocked from transmission outside its predefined communication schedule.

5.3 Testing Self-X Properties

We want the time-triggered organic computing to exhibit also the self-x properties. By creating artificial DNA files, that describe embedded systems in the automotive domain, and feeding these files into the TTOC simulator, it is possible to test the self-building property. Furthermore, with the AHS properties residing in the simulator the self-organization and self-configuration of the middleware can be examined. Here, we are particularly interested in the case of failures of ECUs, if the functionalities of the ECU will be transferred to the other ones, and in the system reconfiguration. In the simulator, it is possible to create timed events that denote ECU failures. By implementing this feature, we observed that right after the ECU was in a failure state, the system reconfigured and reorganized itself, with all the tasks distributed to the remaining ECUs.

6 Conclusion and Future Work

In this paper, we propose the building of a scheduling algorithm that will serve to calculate table schedules during run-time for the newly proposed time-triggered organic computing architecture. TTOC architecture is based on the combination of self-x properties residing in ADNA and AHS with time-triggered techniques to improve the reliability, determinism and safety of embedded systems in autonomous driving vehicles. The TTOC architecture enhances the temporal predictability of the system by ensuring that all tasks are executed at specific times. In addition, the simulation results performed on the TTOC simulator showed that the functionalities of ADNA and AHS remained unchanged.

For our future work, we are planning on testing the communication network between ECUs in the network simulator called OMNet++. The selected time-triggered network is Time Sensitive Network (TSN). The interaction between the network simulator and TTOC process will be handled by a new designed co-simulation controller. Furthermore, we want to test our new architecture with an autonomous driving vehicle simulator like CARLA. In this case we can test our architecture with real autonomous driving application tasks.

Acknowledgment

This work was supported by research project SelfAutoDOC funded by the German Federal Ministry for Economic Affairs and Climate Action (BMWK).

Figure 7 BMWK Logo

References

[1] Richard Anthony, DeJiu Chen, Martin Törngren, Detlef Scholle, Martin Sanfridson, Achim Rettberg, Tahir Naseer, Magnus Persson, and Lei Feng. Autonomic middleware for automotive embedded systems. *Autonomic Communication*, pages 169–210, 2009.

[2] Richard Anthony, Paul Ward, DeJiu Chen, Achim Rettberg, James Hawthorne, Mariusz Pelc, and Martin Tørngren. A middleware approach to dynamically configurable automotive embedded systems. 5 2010.

[3] Jürgen Branke, Moez Mnif, Christian Müller-Schloer, Holger Prothmann, Urban Richter, Fabian Rochner, and Hartmut Schmeck. Organic computing–addressing complexity by controlled self-organization. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 185–191. IEEE, 2006.

[4] Uwe Brinkschulte. An artificial dna for self-descripting and self-building embedded real-time systems. *Concurrency and Computation: Practice and Experience*, 28(14):3711–3729, 2016.

[5] Uwe Brinkschulte. Technical report: Artificial dna-a concept for self-building embedded systems. *arXiv preprint arXiv:1707.07617*, 2017.

[6] Uwe Brinkschulte, Eric Hutter, and Felix Fastnacht. Adapting the concept of artificial dna and hormone system to a classical autosar environment. In *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 35–42. IEEE, 2019.

[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] DFG Schwerpunktprogramm 1183 Organic Computing, 2005–2011. https://gepris.dfg.de/gepris/projekt/5472210.

[9] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[10] Timo Kisselbach, Simon Meckel, Mathias Pacher, Uwe Brinkschulte, and Roman Obermaisser. Organic computing to improve the dependability of an automotive environment. In *International Conference on Architecture of Computing Systems*, pages 211–225. Springer, 2022.

[11] Mario Qosja, Simon Meckel, and Roman Obermaisser. Simulator for time-triggered organic computing. *Procedia Computer Science*, 220:127–134, 2023.

[12] Urban Richter, Moez Mnif, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck. Towards a generic observer/controller architecture for organic computing. In Christian Hochberger and Rüdiger Liskowsky, editors, *INFORMATIK 2006 – Informatik für Menschen, Band 1*, pages 112–119, Bonn, 2006. Gesellschaft für Informatik e.V.