Applying the Concept of Artificial DNA and Hormone System to a Low-Performance Automotive Environment

Uwe Brinkschulte[1(✉)] and Felix Fastnacht[2]

[1] Institut für Informatik, Goethe Universität Frankfurt am Main, Frankfurt, Germany
brinks@es.cs.uni-frankfurt.de
[2] Intedis GmbH & Co. KG, Würzburg, Germany
Felix.Fastnacht@intedis.com

Abstract. Embedded systems are growing very complex because of the increasing chip integration density, larger number of chips in distributed applications and demanding application fields e.g. in autonomous cars. Bio-inspired techniques like self-organization are a key feature to handle the increasing complexity of embedded systems. In biology the structure and organization of a system is coded in its DNA, while dynamic control flows are regulated by the hormone system. We adapted these concepts to embedded systems using an artificial DNA (ADNA) and an artificial hormone system (AHS). Based on these concepts, highly reliable, robust and flexible systems can be created. These properties predestine the ADNA and AHS for the use in future automotive applications.

However, computational resources and communication bandwidth are often limited in automotive environments. Nevertheless, in this paper we show that the concept of ADNA and AHS can be successfully applied to an environment consisting of low-performance automotive microcontrollers interconnected by a classical CAN bus.

Keywords: Artificial DNA · Artificial hormone system · Self-organization · Automotive environment · CAN bus

1 Introduction

Embedded systems are growing very complex because of the increasing chip integration density, larger number of chips in distributed applications and demanding application fields e.g. in autonomous cars. Bio-inspired techniques like self-organization are a key feature to handle the increasing complexity of embedded systems. In biology the structure and organization of a system is coded in its DNA, while dynamic control flows are regulated by the hormone system. We adapted these concepts and developed the Artificial DNA (ADNA) by which

M. Schoeberl et al. (Eds.): ARCS 2019, LNCS 11479, pp. 87–99, 2019.
https://doi.org/10.1007/978-3-030-18656-2_7

the blueprint of the structure and organization of an embedded systems can be described. The ADNA can be stored in every processor of the system (like the biological DNA is stored in every cell of an organism). The tasks described by the ADNA are distributed to the processors in a self-organizing way by an artificial hormone system (AHS). The combination of ADNA and AHS allows to create very robust and flexible systems providing so-called self-X features like self-configuration, self-optimization and self-healing. We have already demonstrated these features in previous publications [8] using an autonomous self-balancing robot vehicle (see e.g. a video in [5]).

In this publication we investigate the applicability of the ADNA and AHS concept to automotive environments. Today's cars are equipped with several processors (electronic control units, ECUs) which perform the tasks necessary to operate the cars' powertrain, safety systems, driving assistants and board entertainment. These systems have to operate at a very high level of robustness and fault-tolerance. So the self-X capabilities of the ADNA and AHS would offer a great potential in this area. However, computational resources and communication bandwidth are often limited in automotive environments. To save costs, ECUs frequently use low-performance microcontrollers with limited computational and memory resources. Furthermore, common automotive bus systems like the CAN bus strictly limit the bandwidth and message sizes.

In the following we show that these limitations can be overcome and the concept of ADNA and AHS can be successfully applied to an environment consisting of low-performance automotive microcontrollers interconnected by a classical CAN bus. Our contribution in this paper is four-fold:

1. We demonstrate the applicability of ADNA and AHS for automotive ECU systems.
2. We compute the memory needs of the ADNA and AHS.
3. We propose an efficient communication scheme for the ADNA and AHS on CAN bus.
4. We evaluate performance measures and the resulting communication and processor load in these systems.

The paper is structured as follows: Related work is presented in Sect. 2. Section 3 describes both the ADNA and the AHS and its application to automotive systems. The adaptation to the target platform of automotive ECUs is presented in Sect. 4. Section 5 shows the evaluation results while Sect. 6 concludes this paper.

2 Related Work

Our approach relies on self-organization in automotive applications. IBM's and DARPAS's Autonomic Computing project [13,15] deals with self-organization of IT servers in networks. Several so-called self-X properties like self-optimization, self-configuration, self-protection and self-healing have been postulated.

The German *Organic Computing* Initiative was founded in 2003. Its basic aim is to improve the controllability of complex embedded systems by using principles found in organic entities [26,27]. Organization principles which are successful in biology are adapted to embedded computing systems.

Self-organization for embedded systems has been addressed especially at the ESOS workshop [4]. Furthermore, there are several projects related to this topic like ASOC [1,21], CARSoC [16,17] or DoDOrg [14]. In the frame of the DoDOrg project, the Artifical Hormone System AHS was introduced [9,14]. [28] describes self-organization in automotive embedded systems. None of these approaches deal with self-organization using DNA-like structures.

DNA Computing [10] uses molecular biology instead of silicon based chips for computation purposes. In [20], e.g. the traveling salesman problem is solved by DNA molecules. In contrast, our approach uses classical computing hardware.

Several authors in [22] emphasize the necessity of redundant processors and sensors in future autonomous cars, however, they do not propose such a fine-grained approach as possible by the ADNA.

In [11] a redundancy scheme for processors in automotive applications is proposed where a voting algorithm is used to determine the validity of results of redundant processors. This is different from our approach which improves the exploit of redundancy using the ADNA.

Our approach relies on classical computing hardware using DNA-like structures for the description and building of the system. This enhances the self-organization and self-healing features of embedded systems, especially when these systems are getting more and more complex and difficult to handle using conventional techniques. Our approach is also different from generative descriptions [12], where production rules are used to produce different arbitrary entities (e.g. robots) while we are using DNA as a building plan for a dedicated embedded system.

To realize DNA-like structures, we have to describe the building plan of an embedded system in a compact way so it can be stored in each processor core. Therefore, we have adapted well known techniques like netlists and data flow models (e.g. the actor model [19]) to achieve this description. However, in contrast to such classical techniques our approach uses this description to build the embedded system dynamically at run-time in a self-organizing way. The description acts like a DNA in biological systems. It shapes the system autonomously to the available distributed multi/many-core hardware platform and re-shapes it in case of platform and environment changes (e.g. core failures, temperature hotspots, reconfigurations like adding new cores, removing cores, changing core connections. etc.). This is also a major difference to model-based [23] or platform-based design [25], where the mapping of the desired system to the hardware platform is done by tools at design time (e.g. a Matlab model). Our approach allows very high flexibility and robustness due to self-organization and self-configuration at run-time while still providing real-time capabilities.

3 Conception of the Artificial Hormone System and DNA

This section briefly describes the concept of the artificial DNA and the underlying artificial hormone system (AHS). For detailed information see [6,7,9].

3.1 Artificial DNA

The approach presented here is based on the observation that in many cases embedded systems are composed of a limited number of basic elements, e.g. controllers, filters, arithmetic/logic units, etc. This is a well known concept in embedded systems design. If a sufficient set of these basic elements is provided, many embedded real-time systems could be completely built by simply combining and parameterizing these elements. Figure 1 shows the general structure of such an element. It has two possible types of links to other elements. The *Sourcelink* is a reactive link, where the element reacts to incoming requests. The *Destinationlink* is an active link, where it sends requests to other elements.

Each basic element is identified by a unique Id and a set of parameters. The sourcelink and the destinationlink of a basic element are compatible to all other basic elements and may have multiple channels.

The Id numbers can be arbitrarily chosen, it is important only that they are unique. Figure 2 gives an example for a PID controller which is often used in closed control loops. This element has the unique Id = 10 and the parameter values for P, I, D and the control period. Furthermore, it has a single sourcelink and destinationlink channel.

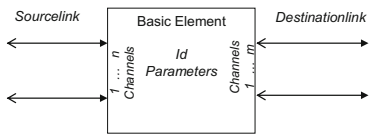

Fig. 1. Structure of a basic element (task)

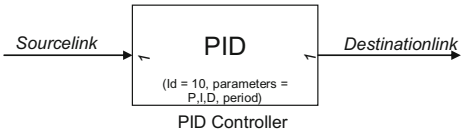

Fig. 2. Sample basic element

Embedded systems can be composed by using these basic elements as building blocks. Figure 3 shows a very simple example of a closed control loop based on

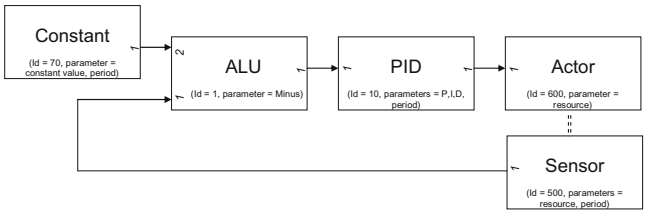

Fig. 3. A closed control loop consisting of basic elements

basic elements. An actor (defined by its resource id, e.g. a motor) is controlled by a sensor (also defined by its resource id, e.g. a speed sensor) applying a constant setpoint value. If we consider the closed control loop to be the *function* of the embedded system, it is divided by the ADNA into *tasks*: the basic elements.

If a sufficient set of standardized basic elements with unique Ids is available, an embedded system will no longer be programmed, but composed by connecting and parametrizing these elements. The building plan of the system can be described by a compact netlist containing the basic elements, its parameters and interconnections. This netlist can be stored in each processor of the system. It therefore represents a digitial artificial DNA (ADNA) which allows to partition and build the system at run-time. Detailed examples and a very memory efficient format to store an ADNA are presented in [6] and [7].

3.2 Building the System from Its ADNA by the AHS

Using the ADNA the system is divided into functions (e.g. control functions, closed control loops, data processing, filtering, etc.) and tasks (the basic elements of a function). Each processor has a local copy of the ADNA and therefore knows all these functions, tasks and their interconnections. It passes this information to the local instance of its artificial hormone system (AHS). The AHS is a completely decentralized mechanism to assign tasks to distributed computing nodes, see [9]. It uses artificial hormones (emulated by short messages) to find the most suitable computing node for each task based on node capability, load and tasks interconnection. It can also detect failing nodes and tasks by missing hormone values. So all basic elements of the ADNA are assigned as tasks at run-time by the AHS to the available processors. These elements are then interconnected according to the ADNA. This means the functions build themselves at runtime in the best possible way on the available processor resources. In case of a processor failure the basic elements are autonomously reassigned and reconnected to other processors as long as there is enough computing power left. Assignment and reassignment of tasks is done in real-time (with a time complexity of $\mathcal{O}(n)$, where n is the number tasks) as proven in [9] and demonstrated by a self-balancing robot vehicle in [7]. The ADNA therefore enables an extremely robust and fine-grain distribution of functions to processors. A function is not bound to a single processor but can be split among several processors on the task (basic element) level. In case of processor failures only the affected basic elements are automatically moved to other processors. Also the importance of basic elements can be derived from the ADNA and used to operate the most important parts if not enough computation power is left to assign all tasks. A detailed description of building a system from the ADNA and complex examples can be found in [7].

3.3 Application of the ADNA and AHS Concept to Automotive Systems

In automotive applications the system functions (anti-locking brake, traction control, stability control, engine control, driving assistants, infotainment, etc.)

are executed by the car's processors, the ECUs. Many of these systems require fail-operational behavior. So a highly robust design is necessary. In classical approaches a function is usually mapped to an ECU (e.g. anti-locking brake to the anti-locking brake ECU). To provide fail-operational behavior, critical ECUs have a backup ECU (1:1 redundancy). In more advanced approaches like e.g. the AutoKonf project [2], several ECUs share a single backup ECU (n:1 redundancy) to reduce the overhead. These approaches apply redundancy on the *function level*. In contrast, the self-healing process of the ADNA and AHS concept provides redundancy on the *task (basic element) level*. This enables the best possible use of the available ECU resources.

If we have e.g. f critical functions, the classical 1:1 redundancy approach requires $2f$ ECUs. Fail-operational behavior can no longer guarantied if 2 or more ECUs fail (the failure of 2 ECUs can disable a function, if the original and the backup ECU are affected). So the fail-operational limit is $\frac{2}{2f} = \frac{1}{f}$. In a 2:1 redundancy approach, $\lceil 3f/2 \rceil$ ECUs are required. Like for the 1:1 approach, fail-operational behavior can no longer be guarantied if 2 or more ECUs fail. However, due to the lower number of ECUs used, the fail-operational ECU limit is better: $\frac{2}{\lceil 3f/2 \rceil}$. In general, the classical n:1 redundancy results in a fail-operational ECU limit of $\frac{2}{\lceil (1+1/n)f \rceil}$.

Using the ADNA/AHS approach, the self-healing mechanism reassigns the tasks of the functions to the remaining ECUs in case of an ECU failure. As long as enough ECUs are available, all functions will stay operational. If we use the same number of $2f$ ECUs for f critical functions like in the classical 1:1 redundancy approach, f ECUs might fail without the loss of a function (since f ECUs are sufficient to execute f functions). So the fail-operation ECU limit is $\frac{f+1}{2f}$. If we use $\lceil 3f/2 \rceil$ ECUs like in the 2:1 approach, this limit calculates to $\frac{\lceil 3f/2 \rceil - f + 1}{\lceil 3f/2 \rceil}$. In general, if we use $e \geq f$ ECUs, the fail-operational limit calculates to $\frac{e-f+1}{e}$. Figure 4 compares the fail-operational limits for different approaches and different number of functions. It can be seen that from this theoretical point of view the ADNA/AHS approach clearly outperforms the classical solutions. Furthermore, in current safety-critical automotive applications usually a fail-safe state is entered if one more failure would lead to a critical event. For the classical redundancy approaches shown above this happens after 1 ECU failure. For the ADNA/AHS approach this happens not before $e - f$ failures. Therefore, it seems reasonable to apply the ADNA/AHS concept to the automotive area. The major question is if the available computational, memory and bandwidth resources are sufficient there to operate this concept. This will be investigated in the next sections.

4 Adaptation to the Target Platform

As target platform we have chosen the Renesas μPD70F35XX microcontroller family [24]. This family contains a dual lockstep V850E2 32 bit processor core and is a common controller for safety-critical ECUs. It is e.g. also used for the

AutoKonf project [2] mentioned above. The controller offers a classical CAN bus interface [3], which is a frequently used communication bus in automotive systems. Table 1 shows key features of the family. The main bottleneck is the low amount of data memory, together with the limited bandwidth and message size of the CAN bus. Clock frequency and program memory are less critical since the ADNA/AHS requires low computational resources and has a small program memory footprint [8].

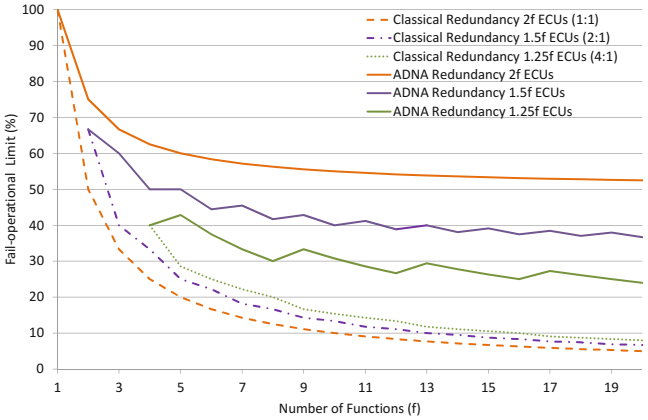

Fig. 4. Fail-operational limits of different redundancy configurations

Table 1. μPD70F35XX microcontroller family key features

Controller:	μPD70F3504	μPD70F3506	μPD70F3508
Data memory:	24 kBytes	40 kBytes	80 kBytes
Program memory:	384 kBytes	512 kBytes	1024 kBytes
Data flash:	32 kBytes	32 kBytes	32 kBytes
Max. clock frequency:	80 MHz	80 MHz	160 MHz

The ADNA/AHS system is completely written in Ansi C 99 and therefore could be easily compiled for the target platform using a GreenHill C compiler for this microcontroller family. Mainly, two modules had to be adapted:

- *AHSBasicOSSupport*, which implements the basic multithreading and synchronization mechanisms for the AHS and ADNA.
- *AHSBasicCommunication*, which implements all basic communication functions for the AHS and ADNA.

In the following sections, we describe these adaptions as well as the resulting data and program memory footprint.

4.1 Basic Operating System Support

This module usually connects the ADNA/AHS system to the operating system. Implementations for Windows and Linux already exist. On this automotive microcontroller target platform, no Windows or Linux support exists. Therefore we have chosen AtomThreads [18] as a basis to implement the AHSBasicOSSupport module. AtomThreads is a lightweight open source multithreaded library, which can be easily adapted to a dedicated microcontroller. AHSBasicOSSupport implements functions to create, terminate, suspend, resume and schedule threads preemptively with different fixed priorities. Furthermore it realizes synchronization functions like mutexes, semaphores, events and timers. To build this efficiently on top of AtomThreads, the AtomThreads library has been extended:

- Functions to suspend, resume and terminate threads have been added.
- Event management has been added.
- To save data memory, the idle thread (which is active when no other thread is ready to run) has been replace by an idle loop. This idle loop does not require its own stack.
- To save more data memory, the initial thread activating the AtomThread scheduler has been turned into a regular thread of the scheduler so it can be further used. In the original implementation this thread is never used again.
- Idle time measurement has been added to the idle loop. This allows to determine the system load.

Overall, using the modified AthomThreads library a very lightweight and efficient basic operating system support module could be built.

4.2 Basic Communication with CAN Bus

The ADNA/AHS system sends messages and hormones via the AHSBasicCommunication module. Hormones are bundled up to a telegram length of 256 Bytes. The maximum length of message telegrams is also 256 Bytes. So the AHSBasiCommunication module has to offer functionality to send and receive telegrams up to that size. The user accessible fields of a classical CAN bus telegram consist of an identifier section of 11 Bits (standard CAN format), a length section of 4 bits and a data section of up to 64 bits (8 bytes). The identifier section also serves for bus-arbitration using a CSMA/CR policy. A logical 0 dominates a logical 1 so as more 0 are in the identifier as higher is the priority of the telegram. To transfer telegrams of up to 256 Bytes via the classical CAN bus, they have to be divided in chunks. We have chosen a division scheme shown in Fig. 5, which is optimized for the format of the hormone and message telegrams. The first byte of these telegrams distinguishes between hormones and messages. Hormone telegrams are broadcasted to all ECUs, while the receiver ECU id of a message telegram is given in the second byte. So we use the 11 bit identifier field to contain the 8 bit sender ECU id, a hormone/message distinction bit (to allow different priorities for hormones and messages) and a 2 bit chunk id to determine

the first chunk of a telegram (10), an even chunk (00), an odd chunk (01) and a last chunk[1] (11). So we can fully use the CAN bus data payload to send and receive hormones and messages, a n byte telegram is divided into $\lceil n/8 \rceil$ chunks. As mentioned above, in case of a message the second byte of the payload of the first chunk indicates the receiver ECU id. Since a sender ECU never sends a new message or hormone telegram before the previous one is completely transmitted, the receiver ECU id of the first chunk can be applied to all following chunks from the same sender ECU id. The distinction of even and odd chunks additionally allows to detect an odd number of lost chunks.

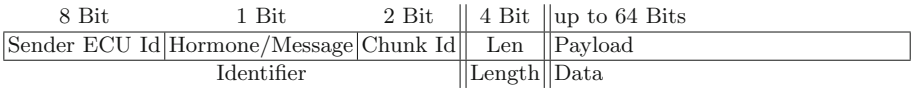

Fig. 5. CAN Bus telegram organization

4.3 Memory Footprint

One of the most critical issues is the low amount of data memory on the target platform. The data memory needs of the ADNA/AHS system can be divided into static and dynamic memory needs. Both could be optimized during the adaption process by e.g. reducing list management overhead, using bit based structures and shrinking oversized buffers. As a result, the dynamic memory needs of the ADNA/AHS could be reduced to:

$$dynMem = 221 + (gt \cdot 32) + (at \cdot 80) + (lt \cdot (96 + mb)) + (rt \cdot 15) + cb + ab \text{ Bytes}$$

with: gt: global number of tasks (basic elements) in the system, at: number of tasks the ECU applies for, lt: number of tasks running on the ECU, rt: number of related tasks, mb: task communication message buffer size, cb: CAN bus communication buffer size, ab: ECU communication message buffer size.

Additionally, AtomThreads need 900 Bytes stack per thread, which is also allocated dynamically. Since we have 2 system threads, the stack memory needs related to the number of running tasks on an ECU calculates to:

$$dynMem_{stack} = 900 \cdot (lt + 2) \text{ Bytes}$$

Finally, when a DNA is read from the data flash memory, administrative memory to operate this DNA is allocated dynamically:

$$dynMem_{DNA} = (dl \cdot 14) + (ln \cdot 4) + ps \text{ Bytes}$$

with: dl: number of DNA lines, ln: number of destination links, ps: parameter size

[1] Only needed if the payload data of a chunk is completely filled, otherwise a length less than 8 bytes indicates the last chunk.

The static data memory needs of the ADNA/AHS system are constant at

$$statMem = 11960 \text{ Bytes}$$

To give a real-number example, an ADNA to realize an anti-locking brake and traction control system[2] requires 16 DNA lines (dl) with 23 destination links (ln) and 210 Bytes parameter space (ps). The resulting number of tasks[3] is 9 (gt). If each ECU in the system applies for all tasks ($at = 9$) and in worst case a single ECU runs all of them ($lt = 9$), each task is related to another task ($rt = 9$) and we have a message buffer size for each task of 128 Bytes (mb), the CAN bus buffer size is 3000 Bytes (cb) and the ECU communication message buffer size is 128 Bytes (ab), the overall data memory needs for this application are:

$$\begin{aligned} data\ memory &= dynMem + dynMem_{stack} + dynMem_{DNA} + statMem \\ &= 6508 + 9900 + 582 + 11960 = 28950 \text{ Bytes} \end{aligned}$$

This easily fits the two bigger controllers of the family (μPD70F3506 and μPD70F3508), see Table 1. For the smallest controller (μPD70F3504) it is a bit too much. However, a major part of the data memory is consumed by the thread stack. So the smallest controller could run 4 tasks at maximum. Due to the dynamic nature of the AHS (a memory overflow automatically produces a suppressor hormone which reduces the number of running tasks on an ECU) the system would autonomously adapt to this situation. This enables the use of the smallest controller if enough are present.

The program memory footprint of the entire ADNA/AHS system is 138 kBytes. So this easily fits all three controllers. Please note that this includes all basic elements, the application itself does not require any additional program and data memory. The running application is stored via the DNA in the data memory using $dynMem_{DNA}$ bytes as calculated above. Also the data flash memory (32 kBytes) used to persistently store different DNAs is by far large enough for a big number of DNAs.

5 Evaluation Results

For the evaluation we have chosen the mid-size controller μPD70F3506. We have used several DNAs from our self-balancing robot vehicle (Balance, BalanceAGV, BalanceFollow, BalanceTurn) as well as two experimental automotive DNAs realizing an anti-locking brake plus traction control (AntiLockTraction) and an anti-locking brake plus traction and cruise control (AntiLockTrCruise). Three different configurations were used: (1) A single ECU was interconnected with the environment via CAN bus. (2) Two ECUs were interconnected to each other and the environment via CAN bus. (3) Two ECUs were interconnected via CAN bus, two more virtual ECUs (on a Windows PC) were interconnected via UDP

[2] Experimental AntiLockTraction DNA from Sect. 5.
[3] Not necessarily all DNA lines require a task, e.g. actor lines.

and UDP/CAN was interconnected by a hub. The results are given in Table 2. The table shows the resulting CAN bus load (at 1 MHz CAN bus frequency) and the computational load of the most occupied real (not virtual) ECU. The hormone cycle time used was 50 ms and the fastest message cycle time was 15 ms. It can be seen that neither the CAN bus load nor the ECU load exceeds critical bounds.

Table 2. Evaluation results

DNA	1 × CAN (1)		2 × CAN (2)		2 × (CAN + UDP) (3)	
	CAN load	ECU load	CAN load	ECU load	CAN load	ECU load
Balance	21%	9%	21%	6%	10%	3%
BalanceAGV	26%	12%	26%	10%	15%	5%
BalanceFollow	28%	13%	28%	10%	23%	8%
BalanceTurn	28%	12%	28%	10%	23%	7%
AntiLockTraction	40%	14%	40%	12%	37%	9%
AntiLockTrCruise	45%	18%	46%	15%	31%	10%

6 Conclusion

In this paper we have shown that it is possible to apply the self-organizing ADNA/AHS concept to an automotive environment with low performance microcontrollers and a classical CAN bus. Due to its self-healing capabilities, this approach can contribute to improve the fail-operational behavior and flexibility of automotive systems. Its failure robustness exceeds traditional approaches. In future, more powerful controllers and busses (like e.g. CAN-FD) will even increase the potential of the ADNA/AHS concept.

In the work presented we have used a modified AtomThreads OS and a proprietary CAN bus protocol. As next step we are investigating the possibility to adapt this concept also to a pure automotive OS like classical AUTOSAR and an AUTOSAR compliant use of the CAN bus. This is challenging due to the static nature of classical AUTOSAR. However, first experiments made using e.g. thread pools show these limitations can be overcome. This would add a completely new quality to AUTOSAR.

References

1. Bernauer, A., Bringmann, O., Rosenstiel, W.: Generic self-adaptation to reduce design effort for system-on-chip. In: IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), San Francisco, USA, pp. 126–135 (2009)
2. BMBF: Autokonf projekt. http://autokonf.de/
3. Bosch: CAN Specifications Version 2.0. http://esd.cs.ucr.edu/webres/can20.pdf

4. Brinkschulte, U., Müller-Schloer, C., Pacher, P. (eds.): Proceedings of the Workshop on Embedded Self-Organizing Systems, San Jose, USA (2013)
5. Brinkschulte, U.: Video of the KDNA controlled robot vehicle. http://www.es.cs.uni-frankfurt.de/index.php?id=252
6. Brinkschulte, U.: An artificial DNA for self-descripting and self-building embedded real-time systems. Pract. Exp. Concurr. Comput. **28**, 3711–3729 (2015)
7. Brinkschulte, U.: Prototypic implementation and evaluation of an artificial DNA for self-describing and self-building embedded systems. In: 19th IEEE International Symposium on Real-time Computing (ISORC 2016), York, UK, 17–20 May 2016
8. Brinkschulte, U.: Prototypic implementation and evaluation of an artificial DNA for self-descripting and self-building embedded systems. EURASIP J. Embed. Syst. (2017). https://doi.org/10.1186/s13639-016-0066-2
9. Brinkschulte, U., Pacher, M., von Renteln, A.: An artificial hormone system for self-organizing real-time task allocation in organic middleware. In: Brinkschulte, U., Pacher, M., von Renteln, A. (eds.) Organic Computing. UCS, pp. 261–283. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-77657-4_12
10. Garzon, M.H., Yan, H. (eds.): DNA 2007. LNCS, vol. 4848. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-77962-9
11. Yi, C.H., Kwon, K., Jeon, J.W.: Method of improved hardware redundancy for automotive system, pp. 204–207 (2015)
12. Hornby, G., Lipson, H., Pollack, J.: Evolution of generative design systems for modular physical robots. In: Proceedings of the IEEE International Conference on Robotics and Automation, ICRA 2001, vol. 4, pp. 4146–4151 (2001)
13. IBM: Autonomic Computing (2003). http://www.research.ibm.com/autonomic/
14. Becker, J., et al.: Digital on-demand computing organism for real-time systems. In: Workshop on Parallel Systems and Algorithms (PASA), ARCS 2006, Frankfurt, Germany, March 2006
15. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **1**, 41–50 (2003)
16. Kluge, F., Mische, J., Uhrig, S., Ungerer, T.: CAR-SoC - towards and autonomic SoC node. In: Second International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2006), L'Aquila, Italy, July 2006
17. Kluge, F., Uhrig, S., Mische, J., Ungerer, T.: A two-layered management architecture for building adaptive real-time systems. In: Brinkschulte, U., Givargis, T., Russo, S. (eds.) SEUS 2008. LNCS, vol. 5287, pp. 126–137. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87785-1_12
18. Lawson, K.: Atomthreads: open source RTOS, free lightweight portable scheduler. https://atomthreads.com/
19. Lee, E., Neuendorffer, S., Wirthlin, M.: Actor-oriented design of embedded hardware and software systems. J. Circ. Syst. Comput. **12**, 231–260 (2003)
20. Lee, J.Y., Shin, S.Y., Park, T.H., Zhang, B.T.: Solving traveling salesman problems with dna molecules encoding numerical values. Biosystems **78**(1–3), 39–47 (2004)
21. Lipsa, G., Herkersdorf, A., Rosenstiel, W., Bringmann, O., Stechele, W.: Towards a framework and a design methodology for autonomic SoC. In: 2nd IEEE International Conference on Autonomic Computing, Seattle, USA (2005)
22. Maurer, M., Gerdes, J.C., Winner, B.L.H.: Autonomous Driving - Technical, Legal and Social Aspects. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-48847-8
23. Nicolescu, G., Mosterman, P.J.: Model-Based Design for Embedded Systems. CRC Press, Boca Raton, London, New York (2010)

24. Renesas: V850E2/Px4 user manual. http://renesas.com/
25. Sangiovanni-Vincentelli, A., Martin, G.: Platform-based design and software design methodology for embedded systems. IEEE Des. Test **18**(6), 23–33 (2001)
26. Schmeck, H.: Organic computing - a new vision for distributed embedded systems. In: 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005), pp. 201–203. Seattle, USA, May 2005
27. VDE/ITG (Hrsg.): VDE/ITG/GI-Positionspapier Organic Computing: Computer und Systemarchitektur im Jahr 2010. GI, ITG, VDE (2003)
28. Weiss, G., Zeller, M., Eilers, D., Knorr, R.: Towards self-organization in automotive embedded systems. In: González Nieto, J., Reif, W., Wang, G., Indulska, J. (eds.) ATC 2009. LNCS, vol. 5586, pp. 32–46. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02704-8_4