Brinkschulte *EURASIP Journal on Embedded Systems* (2017) 2017:23
DOI 10.1186/s13639-016-0066-2

EURASIP Journal on Embedded Systems

RESEARCH

Open Access

Prototypic implementation and evaluation of an artificial DNA for self-descripting and self-building embedded systems

Uwe Brinkschulte

Abstract

Embedded systems are growing more and more complex because of the increasing chip integration density, larger number of chips in distributed applications, and demanding application fields (e.g., in cars and in households). Bio-inspired techniques like self-organization are a key feature to handle this complexity. However, self-organization needs a guideline for setting up and managing the system. In biology the structure and organization of a system is coded in its DNA. In this paper we present an approach to use an artificial DNA for that purpose. Since many embedded systems can be composed from a limited number of basic elements, the structure and parameters of such systems can be stored in a compact way representing an artificial DNA deposited in each processor core. This leads to a self-describing system. Based on the DNA, the self-organization mechanisms can build the system autonomously providing a self-building system. System repair and optimization at runtime are also possible, leading to higher robustness, dependability, and flexibility. We present a prototypic implementation and conduct a real-time evaluation using a flexible robot vehicle. Depending on the DNA, this vehicle acts as a self-balancing vehicle, an autonomous guided vehicle, a follower, or a combination of these.

Keywords: Artificial DNA, Prototypic implementation, Evaluation, Real-time, Self-organization, Self-building, Self-description

1 Introduction

Embedded systems are growing more and more complex because of the increasing chip integration density, larger number of chips in distributed applications, and demanding application fields (e.g., in cars and in households). In the near future, it will become reality to have thousands of computing nodes within an embedded system. Bio-inspired techniques like self-organization are a key feature to handle this complexity. We have developed the Artificial Hormone System (AHS) as a decentralized, self-organizing, self-healing, and self-optimizing mechanism to assign tasks to computing nodes of an embedded real-time system. The AHS is able to handle task assignment in complex embedded systems with a large number of processor cores.

However, to do so the AHS needs a blueprint of the structure and organization of the embedded application. This covers the segmentation of the application into tasks, the cooperation and communication between these tasks, the suitability of the processor cores for each of these tasks, etc. Currently, these assignments are done manually by the system developer, but in the future this is no longer feasible for large embedded systems having a large number of cores and tasks.

The idea is to follow again a bio-inspired principle. In biology the structure and organization of a system is coded in its DNA. This can be adopted to embedded systems. The blueprint of the structure and organization of the embedded system will be represented by an artificial DNA. The artificial DNA can be held compact and stored in every processor core of the system (like

Correspondence: brinks@es.cs.uni-frankfurt.de
Institut für Informatik, Johann Wolfgang Goethe Universität Frankfurt, Frankfurt, Germany

the biological DNA is stored in every cell of an organism). This makes the system *self-descripting* and enables a maximum amount of robustness and flexibility. Now, a mechanism like the AHS can transcribe the artificial DNA to set up and operate the embedded system accordingly. All the needed information for such a process like task structure, cooperation, communication, and core suitability can be derived from the artificial DNA. Therefore, the system becomes *self-building* based on its DNA.

In our previous work [1], we have developed a DNA simulator as a proof of concept. Now we present a prototypic implementation which enables real applications. Furthermore, we conduct an evaluation of real-time properties, robustness, and communication and memory use based on a flexible robotic vehicle platform. Depending on the DNA, this vehicle acts as a self-balancing vehicle, an autonomous guided vehicle, a follower, or a combination of these.

The paper is structured as follows: After the introduction and motivation, related work is presented in Section 2. Section 3 describes the basic ideas and conception of the artificial DNA. The prototypic implementation is described in Section 4; Section 5 presents the evaluation results while Section 6 concludes this paper.

2 Related work

Self-organization has been a research focus for several years. Publications like [2] or [3] deal with basic principles of self-organizing systems, e.g., emergent behavior, reproduction, etc. Regarding self-organization in computer science, several projects and initiatives can be listed.

IBM's and DARPAS's Autonomic Computing project [4, 5] deals with self-organization of IT servers in networks. Several so-called self-X properties like self-optimization, self-stabilization, self-configuration, self-protection, and self-healing have been postulated. The MAPE cycle consisting of *M*onitor, *A*nalyze, *P*lan, and *E*xecute was defined to realize these properties. It is executed in the background and in parallel to normal server activities similar to the autonomic nervous system.

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 [6, 7]. Organization principles which are successful in biology are adapted to embedded computing systems. The DFG priority programme 1183 “Organic Computing” [8] has been established to deepen research on this topic.

Self-organizing and organic computing is also followed on an international level by a task force of the IEEE Computational Intelligence Society (IEEE CIS ETTC OCTF) [9]. Several other international research programs have also addressed self-organization aspects for computing systems, e.g., [10, 11].

Self-organization for embedded systems has been addressed especially at the ESOS workshop [12]. Furthermore, there are several projects related to this topic like ASOC [13, 14], CARSoC [15, 16] or DoDOrg [17]. In the frame of the DoDOrg project, the Artificial Hormone System (AHS) was introduced [17, 18]. Another hormone-based approach has been proposed in [19]. Nicolescu and Mosterman [20] describe self-organization in automotive embedded system. None of these approaches deal with self-description or self-building using DNA-like structures.

DNA computing [21] uses molecular biology instead of silicon-based chips for computation purposes. In [22], e.g., the traveling salesman problem is solved by DNA molecules. 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 [23], 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 [24]) 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 [25] or platform-based design [26], 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 DNA

In the following, the basic conception of the proposed approach is explained in detail. It consists of the system composition model, the structure of the artificial DNA, and how a system is built from its artificial DNA.

3.1 System composition model

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. This fact is also exploited in model-driven design, e.g., by constructing a Matlab model. In our approach, we use it to generate a compact description of the targeted embedded system which can be stored in each processor core to serve as a digital artificial DNA. As we show later, this enables the self-building of the system at run-time including an autonomous reaction to failures and changes in the environment. Figure 1 shows the general structure of such a basic element. This structure is influenced by the way a middleware like the AHS handles tasks and provides maximum flexibility as it allows active and reactive, data-driven, and control-driven, uni- and bidirectional flow. A task is basically a service having two possible types of links to other services. The *Sourcelink* is a reactive link, where the task reacts to incoming requests. It consists of the two functions:

Sourcelink	(the task acts as server)
GetRequest	Looking for an incoming request from others
SendResponse	Sending a response to the requester

The *Destinationlink* is an active link, where the task sends requests to other tasks. It consists of the two functions:

Destinationlink	(the task acts as client)
SendRequest	Sending a request to others
GetResponse	Looking for a response to the request

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. Both links can be bidirectional (in Fig. 1 the filled arrowhead indicates the active direction) or unidirectional. For elements with unidirectional links an incoming request on the sourcelink has no direct response but triggers an outgoing request on the destinationlink. In that case the direction is the one given by the filled arrowhead only. Figure 2 gives some examples which all provide a unidirectional dataflow. The Id numbers are arbitrarily chosen here, it is important only that they are unique. So, we see an arithmetic logic unit (ALU) element with the Id = 1 and the parameter defining the requested operation (minus, plus, mult, div, greater, …). The two input operands are given by channels 1 and 2 of the sourcelink whereas the result is provided via a single-channel destinationlink. Such an element is needed for calculations in a dataflow, e.g., a setpoint comparison in a closed control loop. Another element often used in closed control loops is a PID controller. Here, this element has the unique Id = 10 and the parameter values for P, I, D, and the control period. Furthermore, it has a single unidirectional sourcelink and destinationlink channel. Other popular elements in embedded systems have a unidirectional sourcelink or destinationlink only. Examples are

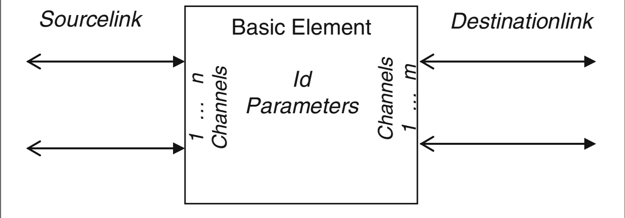


Fig. 1 Structure of a basic element for system description

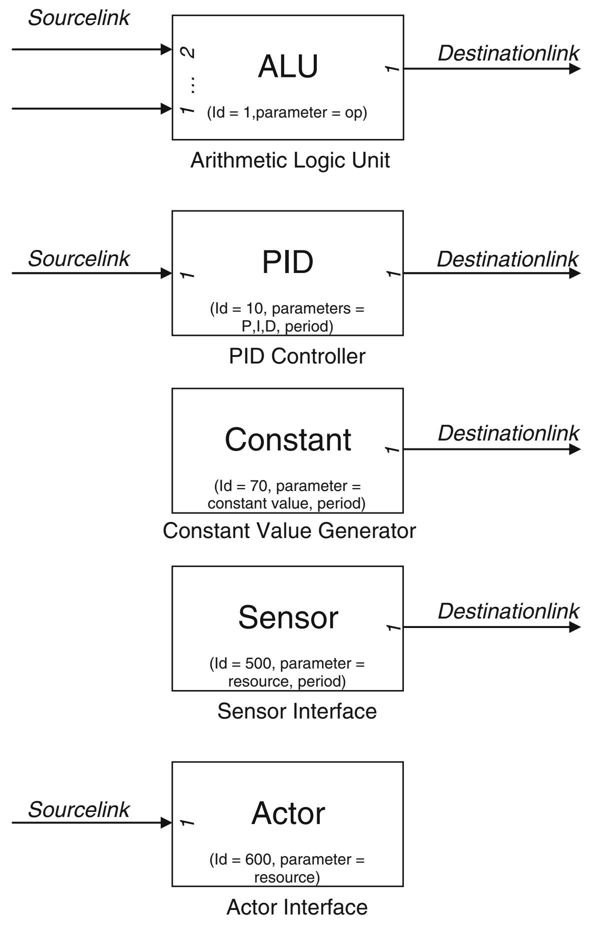


Fig. 2 Sample basic elements with unidirectional links

interfaces to sensors (Id = 500, the parameter resource and period define the specific sensor and its sample period) and actors (Id = 600, resource specifies the specific actor) or a constant value generator (Id = 70, the parameters are the output value produced and its period).

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 the basic elements mentioned above. 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.

3.2 Artificial DNA

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 parameterizing these elements. An example extract of such a set can be found in Section 4. In general, some hundreds of these elements are usually adequate to compose many kinds of embedded real-time systems. Typical elements for different application fields are, e.g., known or can be adapted from model-driven design approaches.

Using such a blueprint, the composition of an embedded system can be stored in a very compact way representing a kind of digital artificial DNA, since it can be used to completely build up the system at run-time. Furthermore, this DNA will be even small enough for complex systems (see Section 5) to be stored in each processor core like the biological DNA is stored in each cell. In this way the embedded system becomes *self-describing*. To create the artificial DNA, the blueprint is transformed into a netlist of basic elements. Each line of the artificial DNA contains the Id of a basic element, its connection to other basic elements (by defining the corresponding destinationlinks for each sourcelink of the basic element) and its parameters:

Artificial DNA line = [Id Destinationlink Parameters]

The destinationlink description in an artificial DNA line can be defined as the following set:

(Destinationlinkchannel:Destination.Sourcelinkchannel...)

Here, *Destinationlinkchannel* gives the channel number of the destinationlink, *Destination* refers to the line of the basic element the destinationlink channel is connected to and *Sourcelinkchannel* is the channel number of the sourcelink channel of the destination element. As an example, the destinationlink description *(1:10.1 1:9.2 2:7.1)* defines that channel 1 of the destinationlink is connected to the sourcelink channel 1 of the basic element in line 10 of the DNA (*1:10.1*) and to the sourcelink channel 2 of the basic element in line 9 (*1:9.2*) while channel 2 of the destinationlink is connected to the sourcelink channel 1 of the basic element in line 7 (*2:7.1*). Figure 4 shows the DNA of the system from Fig. 3 enriched with comments (defined by //). More examples and a very memory efficient format to store a DNA can be found in [1].

Even in case of a very special embedded system not being able to be composed from the set of standardized basic elements, special Ids for user/application specific elements can be defined to solve this issue.

3.3 Building the system from its artificial DNA

Using the artificial DNA, the system now becomes *self-building* at run-time. The DNA serves as the basis for the middleware layer of the distributed embedded system to setup and connect the system tasks. Figure 5 shows the system architecture using a DNA builder and the AHS as middleware layer. There, we call the processor cores of the distributed system DNA processors. First, the DNA builder parses the DNA and segments the system into tasks. Each instance of a basic element becomes a task in the embedded system. Second, the AHS tries to assign tasks to the most suitable processor cores. Core suitability is indicated by specific hormone levels [18]. With the artificial DNA, the suitability of a processor core for a task can be derived automatically by the DNA builder from the

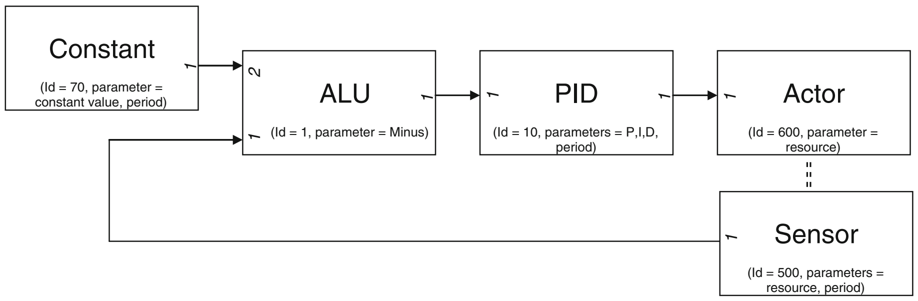

Constant
(Id = 70, parameter = constant value, period)
1
2
ALU
1
1
(Id = 1, parameter = Minus)
1
PID
1
(Id = 10, parameters = P,I,D, period)
1
Actor
(Id = 600, parameter = resource)
1
Sensor
(Id = 500, parameters = resource, period)

Fig. 3 A closed control loop consisting of basic elements

```
1  = 70  (1:2.2) 100 25    // constant setpoint value, period 25 msec
2  = 1   (1:3.1) -         // ALU, control deviation (minus)
3  = 10  (1:4.1) 4 5 6 25  // PID (4, 5, 6), period 25 msec
4  = 600         1         // actor, resource id = 1
5  = 500 (1:2.1) 2 25      // sensor, resource id = 2, period 25 msec
```

Fig. 4 DNA structure of the sample system

Id of the basic element representing the task and the features of the processor core. As an example, a basic element with Id = 10 (PID controller) performs better on a processor core with better arithmetic features while memory is less important. So the appropriate hormone levels can be calculated automatically by the DNA builder and assigned to the AHS. Third, task relationship is also considered for task assignment. The AHS tries to locate cooperating tasks in the neighborhood to minimize communication distances. This has to be indicated also by hormone levels [18]. Using the artificial DNA, task relationship can be derived automatically by the DNA builder from analyzing the destinationlink fields of the DNA lines. This allows to set up the communication links between tasks and to determine cooperating tasks. So the appropriate hormone levels can be generated automatically. All steps of this building process are linear in time with relation to the number of basic elements n, so the overall time complexity is $\mathcal{O}(n)$.

Overall, the artificial DNA represents the blueprint that enables the self-building of a system. In case of failures[1] or changes, the system can be autonomously restored or readapted by the DNA which is present in each DNA processor. This increases system robustness and dependability. The time complexity for restoring or readapting the system is also $\mathcal{O}(n)$, where n is the number of affected basic elements (e.g., the number of basic elements lost by a crash of a DNA processor). The program code for the basic elements used in the DNA can be stored in code repositories distributed in the system. Even changes in the DNA representing changes in the system composition or parameters are possible at run-time providing maximum flexibility.

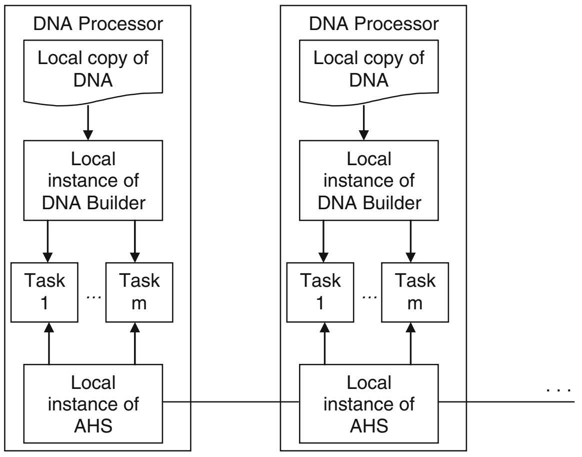


Fig. 5 System architecture

4 Prototypic implementation

As a proof of concept, we first have implemented a simulator for the DNA concept. The results have been published in [1]. This simulator was focused on the ability of self-building a system from its DNA and reconstructing it in case of component failures. So the basic elements were simply dummies in the DNA simulator which are allocated to processor cores, interconnected, and visualized. They provided no real functionality. However, simulation results showed that these basic elements were properly allocated and interconnected by the DNA so self-building and self-repairing is possible.

Encouraged by these promising results, we have decided to implement a real prototype of the DNA concept. In this prototype, the basic elements provide real functionality (e.g., an ALU, a PID controller, etc.) and interaction schemes, so working systems can emerge from a DNA. This allows for a far better evaluation than the simulator does. Communication and memory needs as well as real-time properties can be investigated on a real application example, see Section 5.

Figure 6 shows the detailed architecture of a real DNA processor within the overall system architecture already presented in Fig. 5. The currently active DNA is read from a file by the processor front end consisting of the *DNA Processor Library* and the *DNA Processor Basic Library*. While the first one contains all processor and OS-specific parts, the latter is platform independent and provides generic processor-based functions like retrieving hormone values for basic elements on a given processor core (e.g., an ALU works better on a processor core with strong arithmetic features and therefore deserves higher hormone values to attract this basic element). This is done in cooperation with the *DNA Class Library* which implements all the basic elements. Table 1 shows the basic elements realized in the prototypic implementation of the class library. In addition to the elements already mentioned in the previous section, there are elements to multiplex and demultiplex data values, to limit data values, to define thresholds, switching levels and hysteresis for data values. A complementary filter allows data fusion

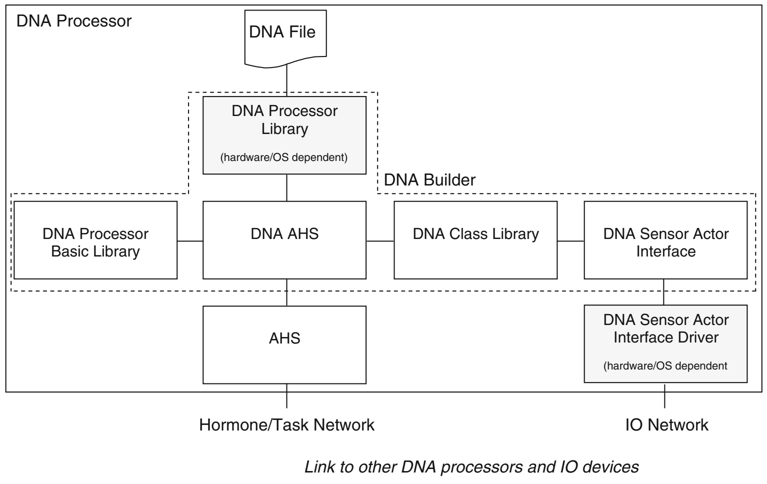


Fig. 6 Architecture of the DNA implementation

similar to but more simple than a Kalmann filter. The DNA checker creates a non-zero output value as soon as the system defined by the given DNA is completely set up and therefore becomes operational on the distributed DNA processors. It can be connected, e.g., to an actor like a LED to indicate the operational state of the system constructed by the DNA (see Fig. 13 in Section 5). The DNA logger writes all input values to a log file and therefore allows the logging of data streams within the system. This small number of basic elements already enables a considerable range of embedded applications, as will be seen in Section 5. All basic elements realized in this prototypic implementation use single precision IEEE float values for data exchange.

The component *DNA AHS* is the connector between all other components and the *AHS Library*. Together these components realize the *DNA Builder* introduced in Section 3.3. Based on the DNA read from file, all necessary basic elements are selected, all interconnections between these basic elements are defined and all hormone values are calculated. This information is promoted to the AHS library which places the basic elements to the DNA processors (see also Section 3.3). To provide an interface to sensors and actors, the *DNA Sensor Actor Interface* component is used. It maps the resource id used as an abstract identification of a specific sensor or actor (see Section 3.1) to the real sensor or actor. This is done in a flexible way by a mapping table allowing the use of various sensors and actors. The *DNA Sensor Actor Driver* component is used to access the real sensor/actor hardware[2]. Only the two components shaded gray in Fig. 6 are platform dependent, all other components are independent from the used processor, sensor/actor hardware, and OS platform. This allows a high portability of the DNA processor implementation. All components are implemented in ANSI C/C++. We compiled them for two target platforms: PC running Windows and Raspberry Pi running Linux. Table 2 shows the memory footprint of both implementations. It can be seen that this footprint is rather small and compact. Only the DNA processor library component for Windows is big.

Table 1 Basic elements implemented

Id	Basic element	Id	Basic element
1	ALU	50	Complementary filter
10	PID	70	Constant
11	P	71	Counter
12	I		
13	D	500	Sensor
		600	Actor
40	Multiplexer		
41	Demultiplexer	997	Stop
42	Level	998	DNA checker
43	Limit	999	DNA logger
44	Hysteresis		
45	Threshold		

Table 2 Components of the DNA implementation

	Raspberry Pi (kB)	Windows PC (kB)
DNA-AHS	31	71
DNA Class Library	34	75
DNA Processor Basic Library	7	19
DNA Processor Library	27	3964
DNA Sensor Actor Interface	41	22
AHS	146	379
Processor overall	270	1662

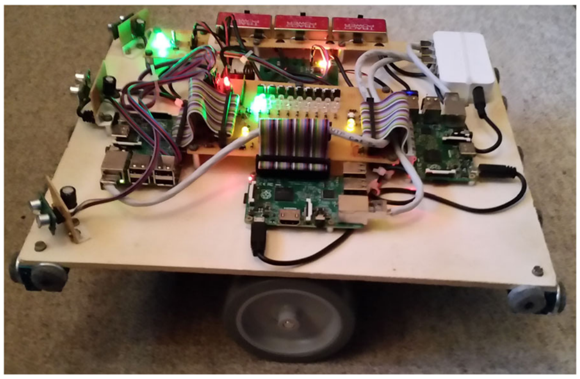


Fig. 7 The demonstrator vehicle

This is due to the fact that Microsoft Foundation Classes (MFC) are used there to show processor information in Windows dialog boxes. The Raspberry Pi implementation uses console IO for this purpose and is therefore rather small. The overall DNA processor size on the Raspberry Pi is only 270 kB. It is possible to create a similar small footprint for Windows PC by using a console version instead of MFC. Both implementations are fully compatible and can be used in a distributed heterogeneous environment.

5 Evaluation

A first evaluation result has already been presented in the previous section. The memory footprint of the artificial DNA implementation is rather small as shown in Table 2. To conduct further evaluations, we have chosen a flexible robot vehicle platform (FOXI)[3] as a demonstrator. This platform can be used either as a self-balancing two-wheel vehicle (as an inverse pendulum, e.g., a Segway personal transporter) or a stable four-wheel vehicle by two additional foldaway supporting wheels. It uses a differential drive and is equipped with various sensors and actors. This allows a wide range of real-time applications. Figure 7 shows a picture of the vehicle (while running a self-balancing application DNA) and Fig. 8 sketches the vehicle architecture. It holds three quadcore Raspberry Pi processors running Raspian Linux. Three cores of each Pi are used as DNA processors resulting in overall 9 DNA processors on the demonstrator platform[4]. The Pis are interconnected by Ethernet. Additionally, a WLAN module is connected. This allows to load DNA files from the outside to the DNA archive of each Pi and to remotely control which DNA is cloned (loaded) from the DNA archive to all DNA processors. It is guaranteed that all DNA processors (the cells) use the same DNA at a time[5]. Furthermore, additional external DNA processors and external sensors and actors, e.g., on a Windows PC can be attached via WLAN to extend the demonstrator[6]. All internal sensors and actors are attached to and shared by the Raspberry Pis via an I^2C bus[7]. Available sensors are three supersonic rangefinders (to detect obstacles left, right, or in front of the vehicle in autonomous driving applications), a three-axis gyroscope and accelerometer (used, e.g., for self-balancing applications), an odometer for the left and the right drive wheel (to measure the distance traveled), and several push buttons as digital inputs. Actors are the left and right motor of the differential drive and several LEDs which can be used as digital outputs or dimmed by PWM. The power supply of each Pi can be shutdown remotely or by a push button to inject a heavy component failure (turning off the power of one Raspberry Pi means the simultaneous hard failure of three DNA processors). Single cores can be shutdown individually as well[8].

Loading a specific DNA to the demonstrator vehicle platform now determines what the vehicle will become. For evaluation purpose, we have created several different DNAs as shown in Figure 9.

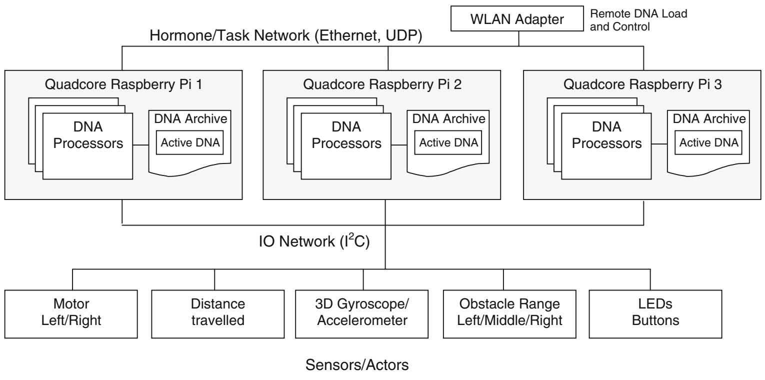


Fig. 8 Architecture of the demonstrator vehicle

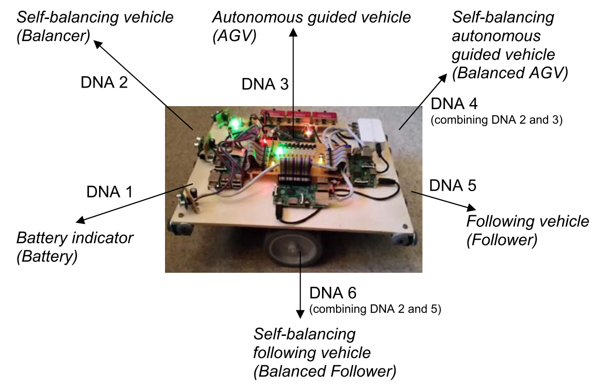

Self-balancing vehicle (Balancer)
Autonomous guided vehicle (AGV)
Self-balancing autonomous guided vehicle (Balanced AGV)
DNA 2
DNA 3
DNA 4 (combining DNA 2 and 3)
DNA 1
DNA 5
Battery indicator (Battery)
Following vehicle (Follower)
DNA 6 (combining DNA 2 and 5)
Self-balancing following vehicle (Balanced Follower)

Fig. 9 Shaping the vehicle by different DNAs

5.1 A first example

As a first very simple example, Fig. 10 shows the block diagram of a DNA for a battery indicator. It displays the battery voltage by a bar of LEDs. For better identification of the basic elements in this figure they are numbered in the left-lower corner. The DNA consists of a battery voltage sensor (basic element 1) which delivers its output each 100 ms to a level discriminator (basic element 3). This discriminator triggers actor LEDs (basic elements 5–12) depending on the input voltage level. The discriminator levels are defined by the parameter set 10 (high voltage), 0.25 (voltage step), and 8 (low voltage). Finally, another LED actor (basic element 4) shows when the application has been completely built or rebuilt (in case of a failure) by its DNA using the DNA checker basic element (2) with a period of also 100 ms. The DNA consist of 12 basic elements and uses only 4 different basic elements. The size of the DNA stored in the compact form proposed in [1] is 162 B.

This simple DNA can be used as a good first example to demonstrate the self-healing properties of the concept in a qualitative way. When loading the DNA and starting the system, the battery indicator builds itself based on its DNA. The upper part of Fig. 11 shows how the battery indicator initially allocates and connects itself to the 9 DNA processors of the 3 Raspberry Pis. The upper picture of Fig. 12 shows a snapshot of the vehicle once the battery indicator has established itself at time $t1$. The bar of red LEDs shows the battery is completely charged. The three yellow LEDs above the bar indicate all 3 Raspberry Pis are powered up and active. At time $t2$ we suddenly shut down the power supply of Raspberry Pi 3. This results in the simultaneous failure of its 3 DNA processors, which hold 3 LED actors (basic elements 5, 6, and 11, see upper part of Fig. 11). So for a very short moment, those 3 red LEDs go dark as can be seen in the middle snapshot picture of Fig. 12. Furthermore, the rightmost yellow LED is now dark since Raspberry Pi 3 is down. However, based on its DNA the battery indicator rebuilds itself autonomously. At time

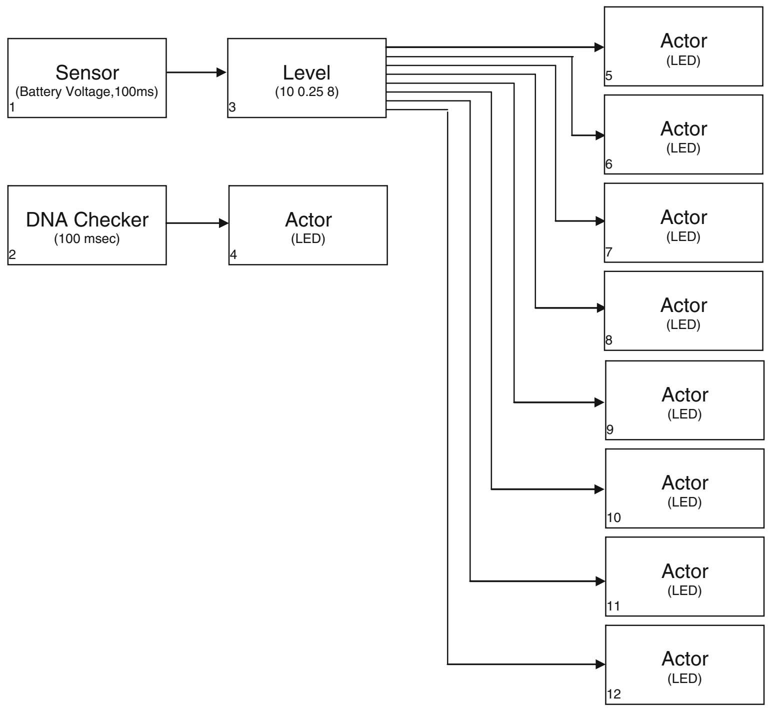


Fig. 10 Block diagram of the battery DNA

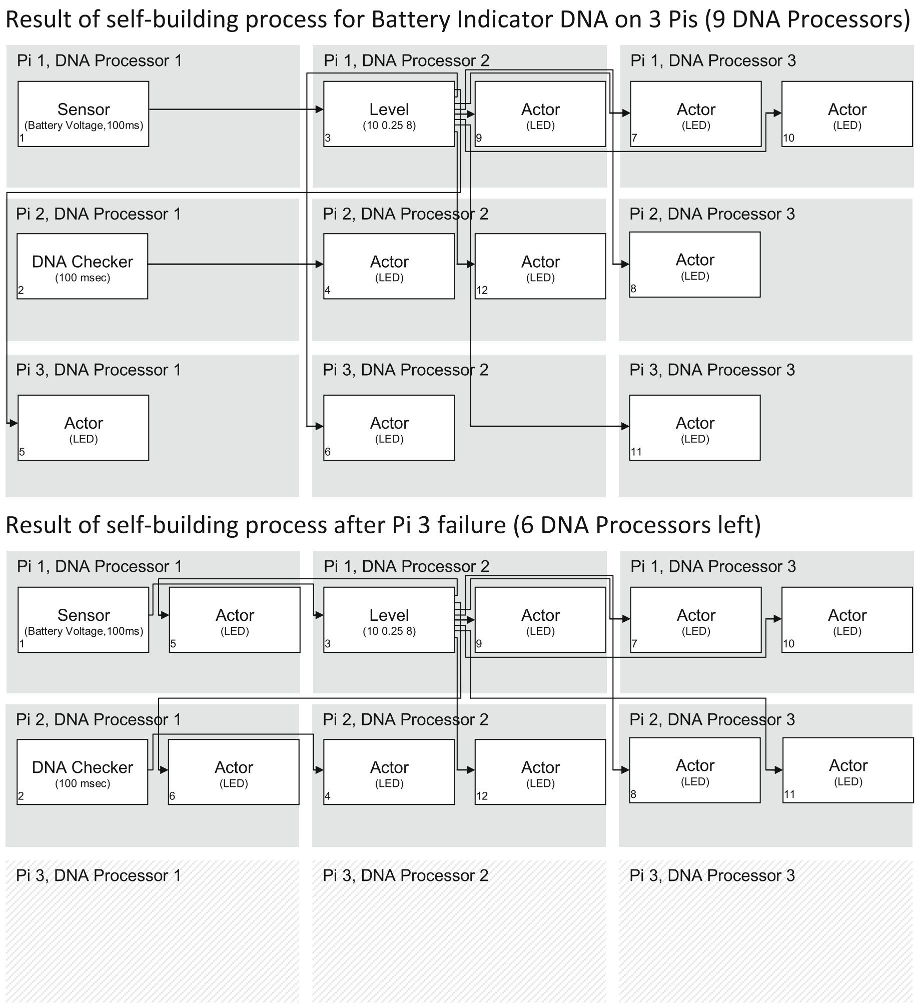


Fig. 11 Allocation and connection of battery indicator basic elements before and after killing of Pi 3

$t3$ the system has completely recovered as can be seen in the lower snapshot picture of Fig. 12. This happens very quickly, as the green LED (basic element 4, located right next to the red LED bar) which checks and indicates the completeness of the DNA every 100 ms, does not even go dark. The lower part of Fig. 11 shows the result of reallocation and reconnection to the remaining 6 DNA processors. Actually, basic elements 5, 6, and 11 have autonomously moved to new DNA processors, as also indicated in Table 3. As mentioned in Section 3.3, the time complexity of this rebuilding process is $\mathcal{O}(n)$. With n equal to 3 basic elements to relocate, this is achieved in less than 150 ms, so the flickering of the LED is too short to be visible.

5.2 More DNAs

In the following, a more detailed quantitative evaluation is conducted based on more complex DNAs. Figure 13 shows a DNA that makes the vehicle self-balancing, e.g., a Segway. Like before, the basic elements in this figure are numbered in the left-lower corner. The self-balancing DNA basically consists of a cascaded closed control loop. The outer loop (basic elements 1, 2, 3, 7, 8, 10, 12, 16) controls the speed of the vehicle by a PID controller. The current speed is determined by differentiating (7, 8) and averaging (10) the odometer data of the left (1) and right (2) wheels of the differential drive. The desired speed setpoint is read by an external sensor (3) via WLAN. A PID controller (16) sets the vehicle angle by the speed deviation (12) with a period of 100 ms. If the vehicle is too slow, it will tilt forward to accelerate. If the vehicle is too fast, it will tilt backward to slow down. With a slight correction regarding the mass center (15, 20), this is the setpoint for the inner loop which controls the vehicle angle (basic elements 4, 5, 9, 11, 14, 18). The current angle is determined by the fusion of accelerometer (4) and gyroscope (5) data

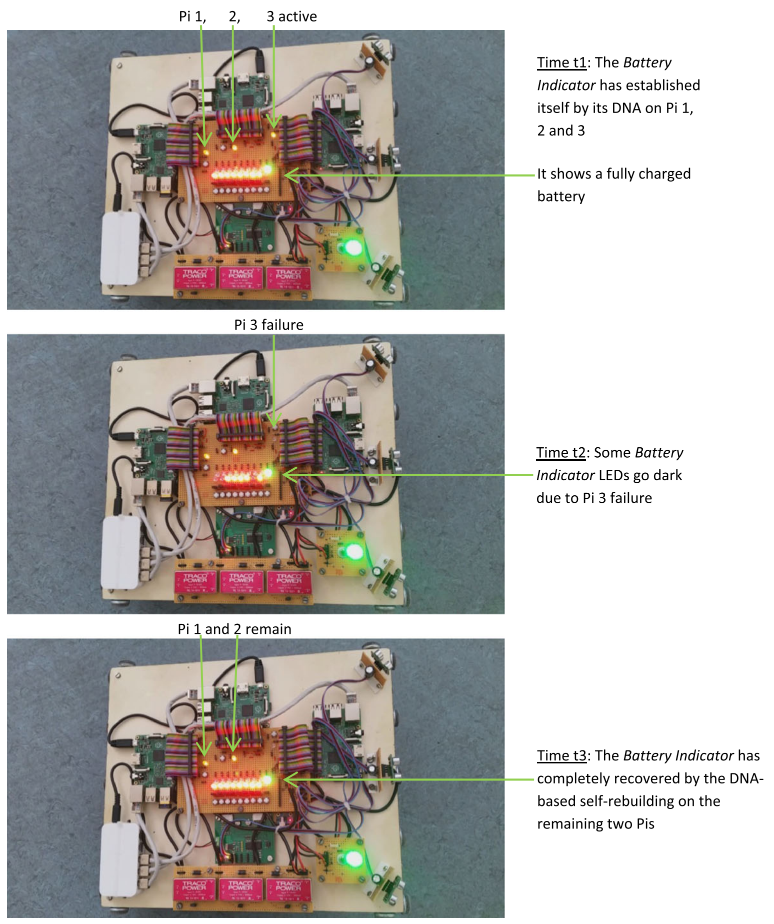

Pi 1, 2, 3 active
Time t1: The *Battery Indicator* has established itself by its DNA on Pi 1, 2 and 3
It shows a fully charged battery
Pi 3 failure
Time t2: Some *Battery Indicator* LEDs go dark due to Pi 3 failure
Pi 1 and 2 remain
Time t3: The *Battery Indicator* has completely recovered by the DNA-based self-rebuilding on the remaining two Pis

Fig. 12 DNA-based self-healing of the battery indicator

using a complementary filter (9). This is necessary because pure accelerometer data is noisy and gyroscope data has a permanent drift. A PID controller (14) accelerates or decelerates the differential drive (18) using the angle deviation (11) with a period of 15 ms to achive the desired angle. The desired direction of the vehicle is read by another external sensor (6) via WLAN and is directly connected to the direction actor (19) of the differential drive. Finally, a LED (17) shows when the application has been completely built or rebuilt (in case of a failure) by its DNA using the DNA checker basic element (13). This DNA consists of 20 basic elements and uses 8 different basic elements. The size of this DNA stored in the compact form proposed in [1] is 188 B. It shows the artifical DNA concept allows a very compact representation of embedded applications.

To further evaluate this, we have created more DNAs: An *Autonomous Guided Vehicle (AGV) DNA* shown in Fig. 14 autonomously drives the vehicle in a maze using the supersonic range finder sensors. This DNA uses the supporting wheels so no self-balancing is necessary. Based on the left and right range finders (basic elements 1, 2), a driving direction is calculated (5, 6, 10, 11, 14, 18). In case of an obstacle very close to the mid range finder (basic element 3), an evasive turn action is provided (7, 12, 15, 16, 19, 20, 22). The vehicle speed is

Table 3 Movement of battery indicator basic elements due to killing of Pi 3

Pi	DNA processor	Basic elements before killing Pi 3	Basic elements after killing Pi 3
1	1	1	1, *5*
1	2	3, 9	3, 9
1	3	7, 10	7, 10
2	1	2	2, *6*
2	2	4, 12	4, 12
2	3	8,	8, *11*
3	1	5	
3	2	6	
3	3	11	

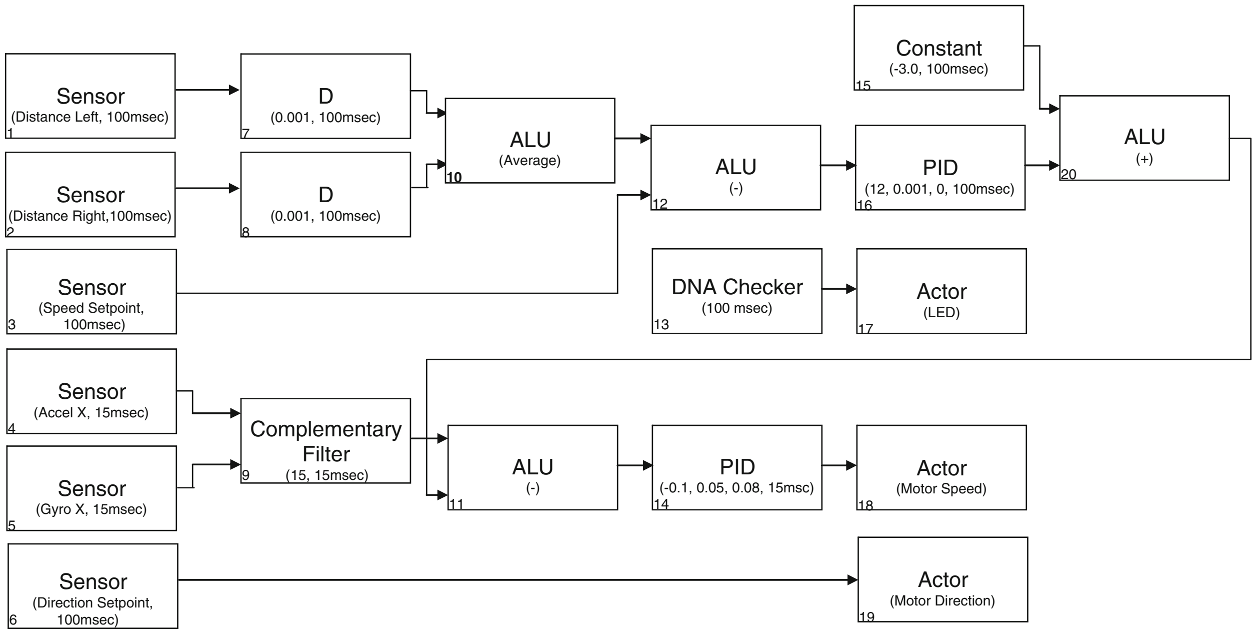

Sensor (Distance Left, 100msec) 1
Sensor (Distance Right,100msec) 2
Sensor (Speed Setpoint, 100msec) 3
Sensor (Accel X, 15msec) 4
Sensor (Gyro X, 15msec) 5
Sensor (Direction Setpoint, 100msec) 6
D (0.001, 100msec) 7
D (0.001, 100msec) 8
Complementary Filter (15, 15msec) 9
ALU (Average) 10
ALU (-) 11
ALU (-) 12
DNA Checker (100 msec) 13
PID (-0.1, 0.05, 0.08, 15msc) 14
Constant (-3.0, 100msec) 15
PID (12, 0.001, 0, 100msec) 16
Actor (LED) 17
Actor (Motor Speed) 18
Actor (Motor Direction) 19
ALU (+) 20

Fig. 13 Block diagram of the balancer DNA

calculated by the lowest value of all three range finders (8, 13, 17). This DNA applies direct control to the left and right motor of the differential vehicle drive (23, 24, 25, 26, 28, 29).

A *Balanced AGV DNA* sketched in Fig. 15 combines the self-balancing (basic elements 1–17) and the AGV DNA (basic elements 18–42) to create a self-balancing autonomous vehicle.

A *Follower DNA* displayed in Fig. 16 lets the vehicle follow an object using the rangefinders. The direction to the obstacle is calculated by the left and right range finders (1, 2, 5, 6, 9, 10, 13, 15, 17, 19, 20). The speed is calculated by a closed PID control loop (14, 16, 21) to keep it at a desired distance of 30 cm (11) from the obstacle. The distance is derived by the minimum of all three range finders (1, 2, 3, 7, 12).

Finally, a *Balanced Follower DNA* shown in Fig. 17 combines the self-balancing (basic elements 1–17) and the Follower DNA (basic elements 18–42) to create a self-balancing follower.

Table 4 gives the sizes of these DNAs. They are all very small and only consist of few different basic elements. Furthermore, the load produced by the communication of the basic elements for each application is given in this table. This load is also considerably small in the range of some kiloBytes per second.

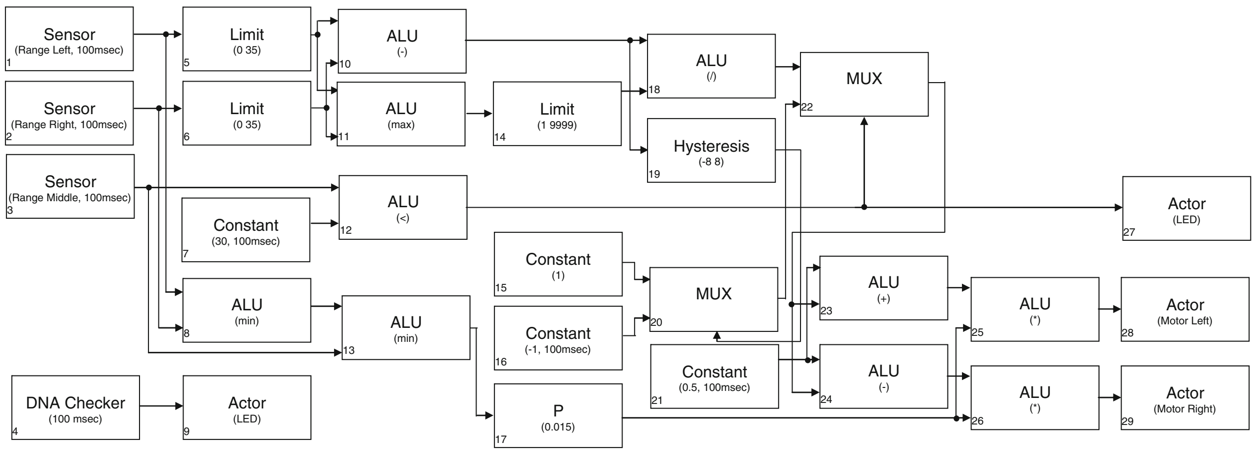
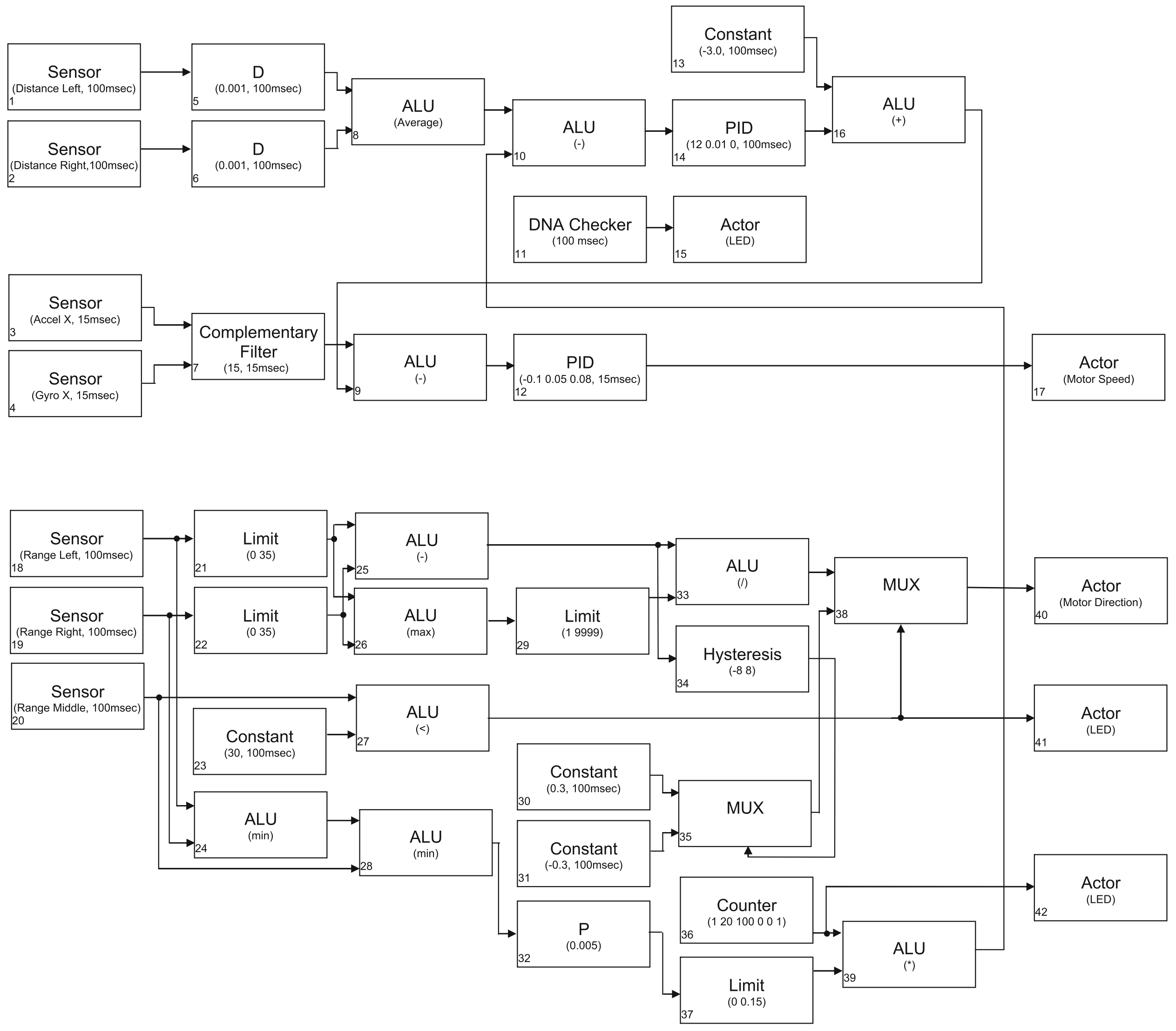


Fig. 14 Block diagram of the AGV DNA



Sensor (Distance Left, 100msec) 1
Sensor (Distance Right,100msec) 2
D (0.001, 100msec) 5
D (0.001, 100msec) 6
ALU (Average) 8
ALU (-) 10
PID (12 0.01 0, 100msec) 14
Constant (-3.0, 100msec) 13
ALU (+) 16
DNA Checker (100 msec) 11
Actor (LED) 15
Sensor (Accel X, 15msec) 3
Sensor (Gyro X, 15msec) 4
Complementary Filter (15, 15msec) 7
ALU (-) 9
PID (-0.1 0.05 0.08, 15msec) 12
Actor (Motor Speed) 17
Sensor (Range Left, 100msec) 18
Sensor (Range Right, 100msec) 19
Sensor (Range Middle, 100msec) 20
Limit (0 35) 21
Limit (0 35) 22
ALU (-) 25
ALU (max) 26
Limit (1 9999) 29
ALU (/) 33
Hysteresis (-8 8) 34
MUX 38
Actor (Motor Direction) 40
Constant (30, 100msec) 23
ALU (<) 27
Actor (LED) 41
Constant (0.3, 100msec) 30
Constant (-0.3, 100msec) 31
MUX 35
ALU (min) 24
ALU (min) 28
P (0.005) 32
Counter (1 20 100 0 0 1) 36
Limit (0 0.15) 37
ALU (*) 39
Actor (LED) 42

Fig. 15 Block diagram of the Balanced AGV DNA

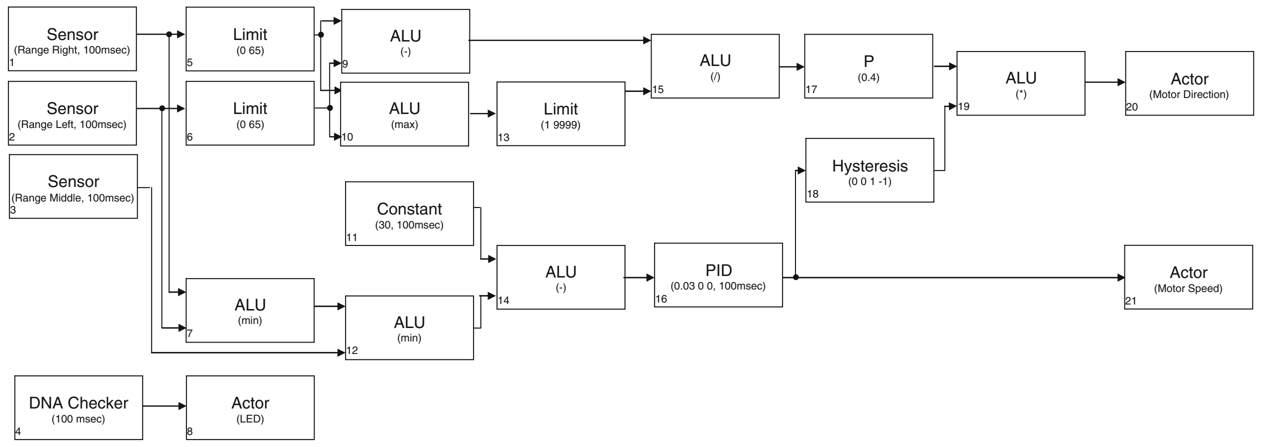
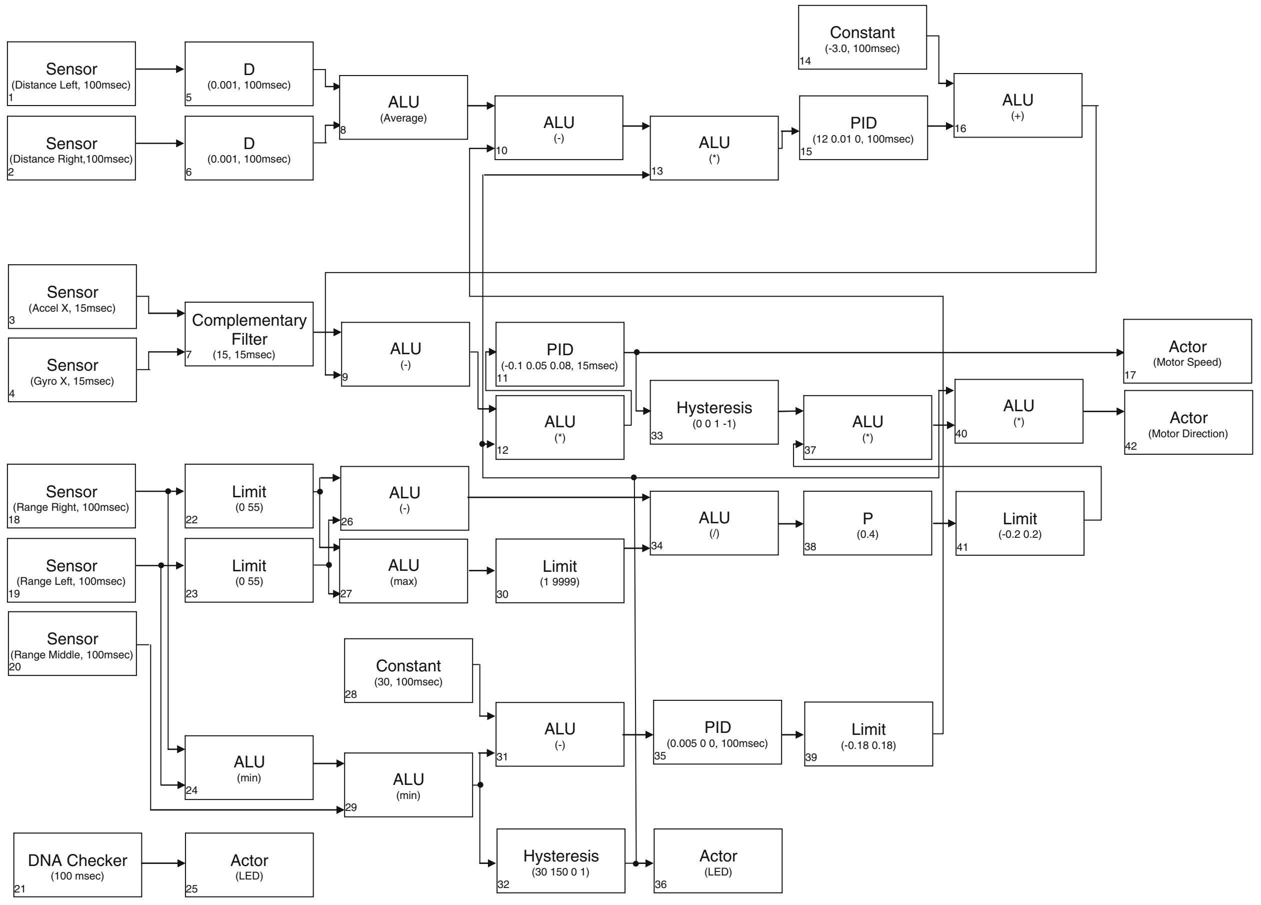


Fig. 16 Block diagram of the Follower DNA



Sensor (Distance Left, 100msec) 1
D (0.001, 100msec) 5
Sensor (Distance Right,100msec) 2
D (0.001, 100msec) 6
ALU (Average) 8
ALU (-) 10
ALU (*) 13
Constant (-3.0, 100msec) 14
PID (12 0.01 0, 100msec) 15
ALU (+) 16
Sensor (Accel X, 15msec) 3
Sensor (Gyro X, 15msec) 4
Complementary Filter (15, 15msec) 7
ALU (-) 9
PID (-0.1 0.05 0.08, 15msec) 11
ALU (*) 12
Hysteresis (0 0 1 -1) 33
ALU (*) 37
ALU (*) 40
Actor (Motor Speed) 17
Actor (Motor Direction) 42
Sensor (Range Right, 100msec) 18
Limit (0 55) 22
ALU (-) 26
ALU (/) 34
P (0.4) 38
Limit (-0.2 0.2) 41
Sensor (Range Left, 100msec) 19
Limit (0 55) 23
ALU (max) 27
Limit (1 9999) 30
Sensor (Range Middle, 100msec) 20
Constant (30, 100msec) 28
ALU (-) 31
PID (0.005 0 0, 100msec) 35
Limit (-0.18 0.18) 39
ALU (min) 24
ALU (min) 29
DNA Checker (100 msec) 21
Actor (LED) 25
Hysteresis (30 150 0 1) 32
Actor (LED) 36

Fig. 17 Block diagram of the Balanced Follower DNA

5.3 Real-time behavior

The basic elements are allocated and connected to the DNA processors by the DNA and the AHS in a self-organizing way. Table 5 shows this exemplarily for the self-balancing DNA. The first row shows the initial allocation to the available 9 DNA processors. After a while, the power supply of Raspberry Pi 2 was shut down turning off three DNA processors simultaneously. The DNA and AHS now reallocate and reconnect the basic elements so the remaining 6 DNA processors can still perform the application. The new allocation is shown in row 2 of the table. The reallocated basic elements are marked in italic.

We use this DNA scenario to evaluate the real-time behavior of the artificial DNA system on the demonstrator platform. The results of the other DNAs are very similar. Figure 18 shows the measured period of the motor control

Table 4 Sample DNAs

DNA	Basic elements	Different basic elements	DNA size (Bytes)	Communication load (Bytes/s)
Battery	12	4	162	2100
Balancer	20	8	188	9513
AGV	29	10	338	7140
Balanced AGV	42	13	536	15,183
Follower	21	10	270	5040
Balanced Follower	42	11	536	22,953

Table 5 Balancer: allocation of basic elements before and after killing of Pi 2

Pi	DNA processor	Basic elements before killing Pi 2	Basic elements after killing Pi 2
1	1	13, 17	*6*, 13, 17, *19*
1	2	4, 5, 9, 11	4, 5, 9, 11
1	3	15, 20	*12*, 15, *16*, 20
2	1	6, 19	
2	2	10, 12, 16	
2	3	3	
3	1	14, 18	*3*, 14, 18
3	2	1, 7	1, 7, *10*
3	3	2, 8	2, 8

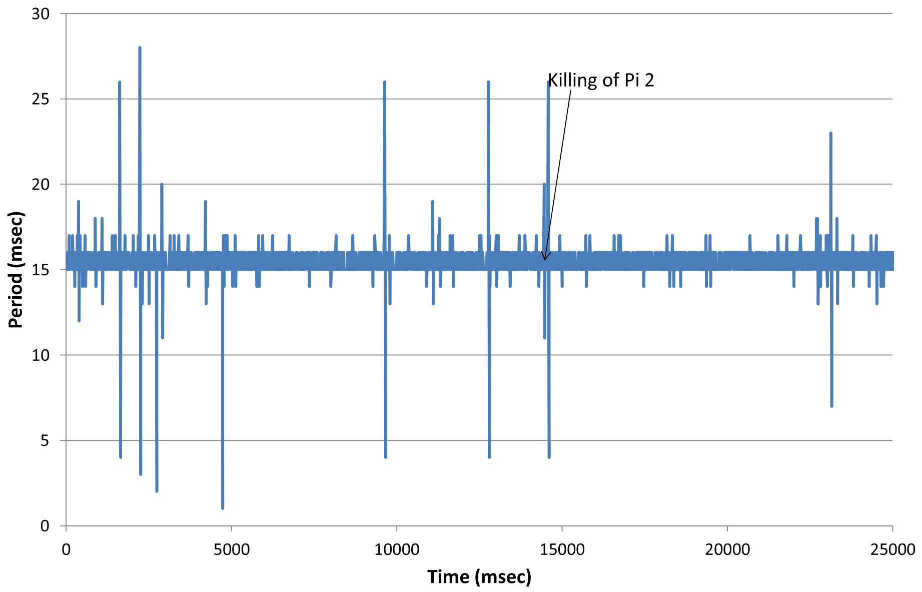


Fig. 18 Real-time properties: period of the motor control signal

signal (output of basic element 14 in the balance DNA). As mentioned above, the period of the inner control loop is set to 15 ms by the DNA parameters. Overall this period is reached quite well; however, Raspian Linux is no real-time operating system. So some occasional spikes in the range of plus/minus 12 ms to the intended period can be observed. It is very interesting that shutting down Pi 2 at timestamp 14,800 only produces a spike of plus/minus 10 ms, which is in the range of the other spikes. So the self-rebuilding of the system by the DNA works almost seamlessly and fast. The vehicle does not lose its balance. This can be seen in Fig. 19. Here, the deviation of the desired speed and angle of the vehicle are shown. After the initial self-building of the system which takes about 700 ms, the target angle and speed are well reached. The small angle deviations of about plus/minus 5° result from the remaining noise of the accelerometer, friction, mechanical play of the drive and the occasional spikes in the control period as shown in the previous diagram. Interestingly, the shutdown of Pi 2 does not cause a major disturbance in

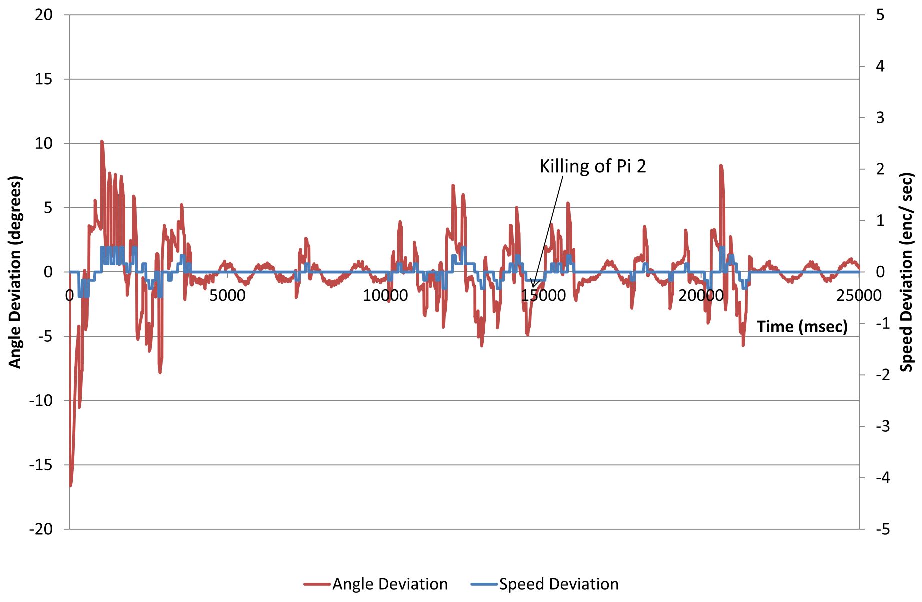


Fig. 19 Angle and speed deviation of the vehicle controlled by the balancer DNA

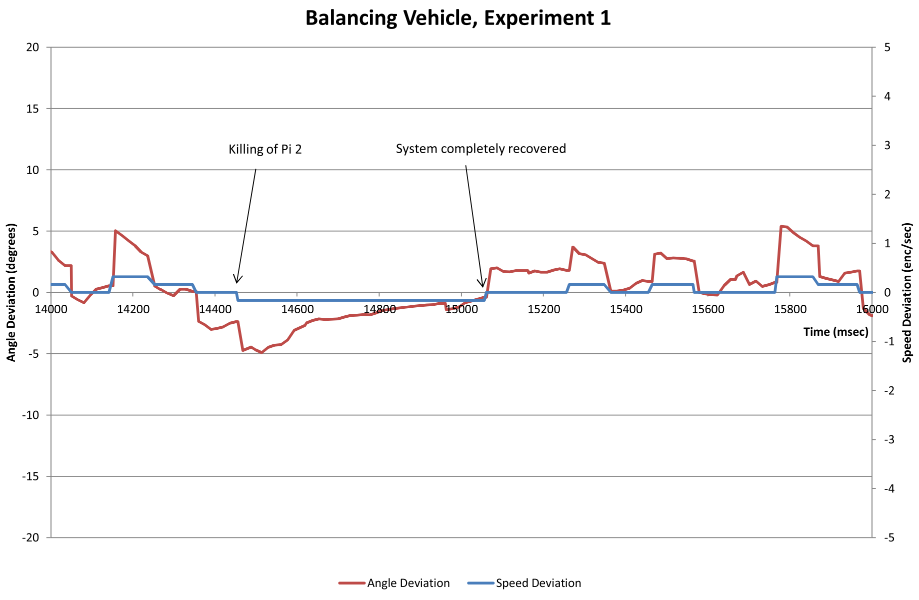


Fig. 20 Extract of Fig. 19

angle and speed control. This is shown in more detail in Fig. 20. The deviation is in the range of the other deviations. The rebuilding process is fast enough to keep well the balance and the speed. Additionally, Fig. 21 shows the motor control signal itself. Also here it can be seen that the shutdown of Pi 2 does not cause any extraordinary change in this signal.

6 Conclusions

In this paper we presented an approach to use a digital artificial DNA for self-describing and self-building systems. This DNA is deposited in each computation node as a blueprint to build and repair the system autonomously at runtime. Mimicking biology this way provides robustness and dependability. The prototypic implementation of the DNA approach enabled an evaluation on a real-world scenario, a robot vehicle. The results showed that the memory and communication overhead of the implementation are rather small, application DNAs are compact, and can be built with a limited number of basic elements. Self-building and self-repairing of the application at runtime can meet real-time requirements.

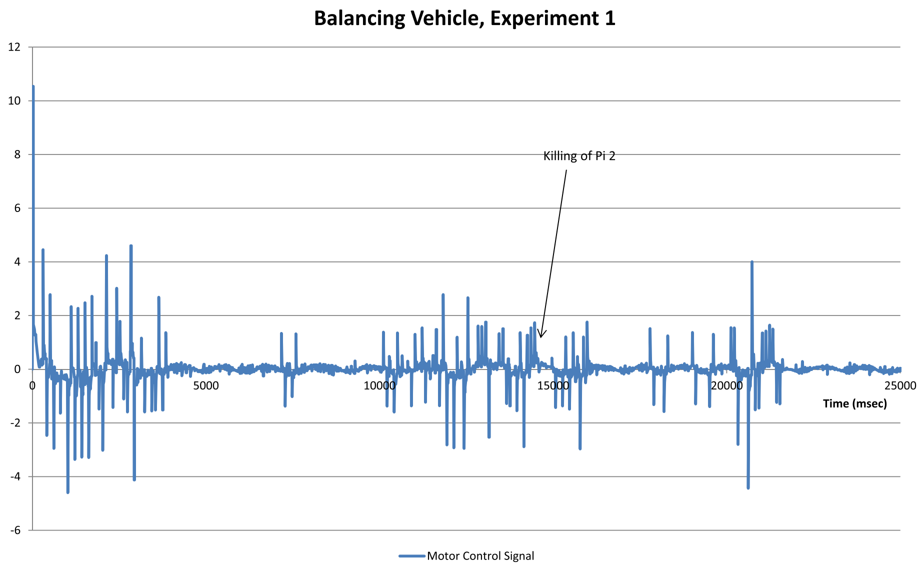


Fig. 21 Motor control signal

Future work will focus on larger DNAs in combination with more DNA processors and dynamic DNA modification at runtime to allow system evolution.

Endnotes

[1]Permanent processor failures like e.g. core crashes, which are detected by missing hormones of the AHS or temporary failures like e.g. single event upsets in memory, which can be detected by monitoring circuits. Communication failures are currently not handled and will be subject of future work.

[2] This driver allows not only access to real sensors and actors, but also to simulated ones. So the implemented DNA system is able to handle mixed environments consisting of real and simulated hardware.

[3] **F**lexible **O**rganix e**X**perimental veh**I**cle

[4] Since Raspian Linux is no real-time OS, the fourth core of each Pi is spared for operating system usage.

[5] However, the DNA on all processors can be changed at run-time simultaneously.

[6] As a restriction, external DNA processors have no access to the internal sensors and actors of the vehicle, so basic elements attached to these sensors and actors will not be allocated to external DNA processors. Furthermore, real-time capabilities of external DNA processors, sensors and actors are limited due to the WLAN connection.

[7] since the native I^2C bus interface of the Raspberry Pi is not multi-master capable, we added some additional hardware support to realize multi-master access.

[8] Failures of the communication system are not in the focus of this demonstrator. This will be covered by future research work.

Competing interests
The authors declare that they have no competing interests.

Received: 11 October 2016 Accepted: 22 December 2016
Published online: 03 February 2017

References

1. U Brinkschulte, An artificial DNA for self-descripting and self-building embedded real-time systems. Concurrency Computat.: Pract. Exper. 2016. **28**, 3711–3729 (2016)
    1. U Brinkschulte, (2016), . 2016; Vol.28, Pages: 3711âĂŞ3729, Wiley
2. G Jetschke, *Mathematik der Selbstorganisation*. (Harry Deutsch Verlag, Frankfurt, 1989)
3. R Whitaker, Self-Organization, Autopoiesis, and Enterprises (1995). http://www.enolagaia.com/RW-ACM95-Main.html
4. IBM, Autonomic Computing (2003). http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf
5. JO Kephart, DM Chess, The Vision of Autonomic Computing. IEEE Comput. **36**(1), 41–50 (2003)
6. VDE/ITG (Hrsg.), VDE/ITG/GI-Positionspapier Organic Computing: Computer und Systemarchitektur im Jahr. GI, ITG, VDE, (2003). https://www.gi.de/fileadmin/redaktion/Presse/VDE-ITG-GI-Positionspapier_20Organic_20Computing.pdf
7. H Schmeck, in *8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005)*. Organic Computing - A New Vision for Distributed Embedded Systems (IEEE, Seattle, 2005), pp. 201–203
8. DFG Schwerpunktprogramm, Organic Computing (2007). http://gepris.dfg.de/gepris/projekt/5472210
9. IEEE, Organic Computing Task Force (2009). http://www.neuroinformatik.rub.de/thbio/project/oc
10. EU, Program Future Emerging Technolgies FET - Complex systems (2009). https://cordis.europa.eu/ist/fet/co.htm
11. CSIRO, Centre for Complex Systems (2009). https://www.natureindex.com/institution-outputs/australia/csiro-centre-for-complex-systems-science-css/563ab1fe140ba0097e8b4574
12. U Brinkschulte, CMüller-Schloer, M Pacher (eds.), *Proceedings of the Workshop on Embedded Self-Organizing Systems* (Usenix, San Jose, 2013)
13. G Lipsa, A Herkersdorf, W Rosenstiel, O Bringmann, W Stechele, in *2nd IEEE International Conference on Autonomic Computing*. Towards a Framework and a Design Methodology for Autonomic SoC (IEEE, Seattle, 2005)
14. A Bernauer, O Bringmann, W Rosenstiel, in *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), San Francisco, USA*. Generic Self-Adaptation to Reduce Design Effort for System-on-Chip, 717 (IEEE, San Francisco, 2009), pp. 126–135
15. F Kluge, J Mische, S Uhrig, T Ungerer, in *Second International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2006)*. CAR-SoC - Towards and Autonomic SoC Node (HiPEAC, L'Aquila, 2006)
16. F Kluge, S Uhrig, J Mische, T Ungerer, in *6th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2008)*. A Two-Layered Management Architecture for Building Adaptive Real-time Systems, (Capri, 2008)
17. J Becker, K Brändle, U Brinkschulte, J Henkel, W Karl, T Köster, M Wenz, H Wörn, in *Workshop on Parallel Systems and Algorithms (PASA), ARCS 2006*. Digital On-Demand Computing Organism for Real-Time Systems (GI, Frankfurt, 2006)
18. U Brinkschulte, M Pacher, Av Renteln, in *Organic Computing*. An Artificial Hormone System for Self-Organizing Real-Time Task Allocation in Organic Middleware (Springer, Heidelberg, 2008)
19. W Trumler, T Thiemann, T Ungerer, in *Biologically inspired Cooperative Computing, IFIP 19th World Computer Congress 2006, August 21-24 2006, Santiago de Chile, Chile*. An Artificial Hormone System for Self-organization of Networked Nodes (IFIP, Santiago de Chile, 2006)
20. G Weiss, M Zeller, D Eilers, R Knorr, in *Autonomic and Trusted Computing, Brisbane, Australia*. Towards Self-organization in Automotive Embedded Systems (Springer, Brisbane, 2009), pp. 32–46
21. MH Garzon, H Yan (eds.), *DNA Computing, 13th International Meeting on DNA Computing, DNA13, Memphis, TN, USA, June 4-8 2007, Revised Selected Papers, volume 4848 of Lecture Notes in Computer Science* (Springer, Heidelberg, 2008)
22. JY Lee, S-Y Shin, TH Park, B-T Zhang, Solving traveling salesman problems with dna molecules encoding numerical values. Biosystems. **78**(1–3), 39–47 (2004)
23. GS Hornby, H Lipson, JB Pollack, in *Robotics and Automation, 2001. Proceedings 2001 ICRA*. IEEE International Conference on. volume 4, Evolution of generative design systems for modular physical robots (IEEE, Seoul, 2001), pp. 4146–4151
24. EA Lee, S Neuendorffer, MJ Wirthlin, in *Journal of Circuits, Systems and Computers*. Actor-Oriented Design Of Embedded Hardware And Software Systems (World Scientific, Singapore, 2003)
25. G Nicolescu, PJ Mosterman, *Model-Based Design for Embedded Systems*. (CRC Press, Boca Raton, London, 2010)
26. A Sangiovanni-Vincentelli, G Martin, in *IEEE Design and Test, Vol 18, No. 6*. Platform-Based Design and Software Design Methodology for Embedded Systems (IEEE, New York, 2001), pp. 23–33