Fault Detection in Autonomous Robots Based on Fault Injection and Learning

Anders Lyhne Christensen, Rehan O'Grady,
Mauro Birattari, and Marco Dorigo

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2007-004

February 2007

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:

IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2007-004

Fault Detection in Autonomous Robots Based on Fault Injection and Learning

Anders Lyhne Christensen, Rehan O'Grady, Mauro Birattari, and Marco Dorigo

IRIDIA, CoDE, Université Libre de Bruxelles,
50, Av. Franklin. Roosevelt CP 194/6,
1050 Bruxelles, Belgium
alyhne@iridia.ulb.ac.be,{rogrady,mbiro,mdorigo}@ulb.ac.be

Abstract. In this paper, we study a model-free approach to fault detection for autonomous robots. Our hypothesis is that hardware faults will change the flow of sensory information and the actions performed by the control program. By detecting these changes, the presence of faults can be inferred. In order to test our hypothesis, we collect data from three different tasks performed by real robots. During each experiment, we record all sensory inputs from the robots while they are operating normally and when in a simulated fault state. Given that faults are simulated and controlled, we can correlate the flow of sensory inputs with the state of the robot. We use a basic machine learning technique, that is, back-propagation neural networks, to synthesize fault detection components for robots performing specific tasks. We show that based on the flow of sensory inputs alone, we achieve good, model-free fault detection for real robots.

1 Introduction

As more and more robots are introduced in space, industry, and private homes, fault detection is becoming an increasingly important issue to address. When a robot stops exhibiting its intended behavior, either due to an internal fault or external factors, it can become a costly and/or a dangerous affair (Isermann, 1997). The problem is often exacerbated if the fault is not detected in a timely manner. In a recent paper (Carlson and Murphy, 2003), the reliability of seven mobile robots from three different manufacturers was tracked over a period of two years and the average mean time between failures (MTBF) was found to be 8 hours. The result suggests that faults in mobile robots are quite frequent. Fault detection can be achieved by adding special-purpose hardware such as torque and position sensors (Terra and Tinos, 2001). Adding additional hardware increases cost and complexity, and it is, therefore, something that we would like to avoid in many cases. The National Aeronautics and Space Administration (NASA) is, for instance, studying deployment of cooperating swarms of hundred to thousands of small-scale autonomous robots to explore the solar system (Hinchey et al., 2004). Given the high number of robots, simplicity and small size are high priorities. Similarly, for domestic adoption of service and leisure robots, the number and complexity of components have to be kept low in order to reach a price point that allows for high market penetration (Kochan, 2005).

In this study, we propose a general method for performing fault detection for autonomous robots. Our method requires no special fault detection hardware and relatively few computational resources to run the fault detection software. It relies on recording sensory data, firstly over a period of time when a robot is operating as intended, and secondly over a period of time when various types of simulated hardware and mechanical faults have been injected. Using knowledge of how the flow of information changes after a fault has occurred, we are able to detect faults.

Technically, a fault is an unexpected change in system function which hampers or disturbs normal operation, causing unacceptable deterioration in performance (Isermann and Ballé, 1997). A *fault tolerant* system is capable of continued operation, possibly at a degraded performance, in the event of faults in some of its parts. Fault tolerance is a sought-after property for critical systems due to economic and/or safety concerns. What we study in this paper is the activity known as *fault detection* in autonomous robots. Fault detection is a binary decision process confirming whether or not a fault has occurred in a system. Other aspects of achieving fault tolerance for a system might include *fault diagnosis*, namely determining the type and location of faults, and *protection* which comprises any steps necessary to ensure continued safe operation of the system (Isermann and Ballé, 1997).

2 Related Work

Fault detection is based on observations of a system's behavior. Deviations from normal behavior can, for instance, be interpreted as symptoms resulting from a fault in the system. A fault detection approach is a concrete method for *observation processing.* A large body of research in *model-based fault detection* approaches exists (Gertler, 1988, Isermann and Ballé, 1997). In model-based fault detection some model of the system or of how it is supposed to behave is constructed. The actual behavior is then compared to the predicted behavior and deviations can be interpreted as symptoms of faults. A deviation is called a *residual*, that is, the difference between the estimated and the observed value. In the domain of mobile autonomous robots, accurate mathematical models are not feasible due to uncertainties in the environments, noisy sensors, and imperfect actuators. A number of methods have been studied to deal with these uncertainties. Artificial neural networks and radial basis function networks have been used for fault detection and identification based on residuals (Vemuri and Polycarpou, 1997, Terra and Tinos, 2001, Patton et al., 2000).

Another popular approach to fault detection is to operate with multiple models concurrently. Each model corresponds to a fault state, for example a broken motor, a flat tire, and so on. The fault corresponding to a particular model is considered to be detected when that model's predictions are a sufficiently close match to the currently observed behavior. Banks of Kalman filters have been used for such state estimation (Roumeliotis et al., 1998, Goel et al., 2000). Kalman filters are based on the assumption that the modelled system can be approximated as a Markov chain built on linear operators perturbed by Gaussian noise (Kalman, 1960). Robotics systems are, like many other real-world systems, inherently nonlinear. Furthermore, discrete fault state changes can result in discontinuities in behavior. Thus, the underlying assumptions do not hold well in the robotics domain and many of these models are, therefore, not generally applicable.

As a potential solution to this issue, dynamic Bayesian networks have been proposed and studied since assumptions about linearity are not required by this technique (Lerner et al., 2000). Recently, computationally efficient approaches for approximating Bayesian belief using *particle filters* have been studied as a means for fault detection and identification (Dearden et al., 2004, Verma et al., 2004, Li and Kadirkamanathan, 2001). Particle filters are Monte Carlo methods capable of tracking hybrid state spaces of continuous noisy sensor data and discrete operation states. The key idea is to approximate the probability density function over fault states given the observed data by a swarm of points or particles. One of the main issues related to particle filters is tracking multiple low-probability events (faults) simultaneously. A scalable solution to this issue has recently been proposed (Verma and Simmons, 2006).

Artificial immune-systems (AIS) represent a biologically inspired approach to fault detection. An AIS is a classifier that distinguishes between *self* and *non-self* (Forrest et al., 1994). In fault detection, "self" corresponds to fault-free operation while "non-self" refers to observations resulting from a faulty behavior. AIS have been applied to robotics, see for example (Canham et al., 2003) in which fault detectors are obtained for a Khepera robot and for a control module of a BAE Systems Rascal robot. The two systems are first trained during fault-free operation and their capacity to detect faults is then tested.

3 Methodology

Several methods for fault detection base classification on the most recent observations only. The approach presented in this study allows classification based on both current and past observations, since many faults can only be detected if a system is observed over some time. This is especially true for mechanical faults in mobile robots; a fault causing a wheel to block, for instance, can

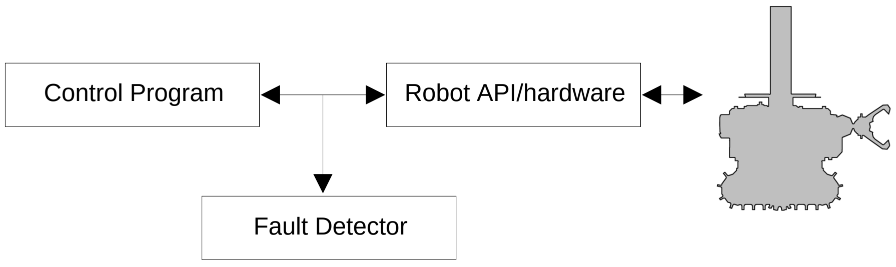


Fig. 1: The fault detection module monitors the sensory data read by the control program and the consequent control signals sent to the actuators. The fault detection module is passive and does not interact with the robot hardware or the control program.

only be detected once the robot has tried to move the wheel for a period of time long enough for the absence of movement to be detectable. This period of time could be anywhere from a few milliseconds if, e.g., dedicated torque sensors in the wheels are used, to seconds if the presence of a fault has to be inferred based on information from non-dedicated and imprecise sensors.

The approach that we suggest in this study shares features with AIS based fault detection in the sense that a classifier discriminates between normal and incorrect behavior without the use of any explicit model. The aim is to detect the difference between normal and abnormal behaviors caused by the occurrence of one or more faults, based on the flow of information from a robot's sensors and the subsequent flow of control signals from the control program to the robot's actuators. The method is independent of the underlying system and learning is based on observations collected when the system operates normally and after faults have been injected.

We focus on detecting faults in a system with a predefined behavior. We assume that the correct behavior has been specified in the form of a control program that steers a robot to perform the task it is intended to perform. The fault detection problem is to determine if the robot performs this task correctly, or if some fault in the hardware or in a software sub-system (but not in the control program itself) is deteriorating the performance. If a fault is detected, a signal can be sent to the control program itself, another robot, or a human operator. In our design, the fault detector is an isolated software component that passively monitors the performance of the robot through the information that flows in and out of the control program. A conceptual illustration of the relationship between the control program, the robotic platform, and the fault detection module can be seen in Fig. 1.

We take a black-box approach and consider only the inputs and the outputs of the control program, that is, the robot's flow of sensory information into the control program and the resulting flow of control signals sent to the actuators. Our hypothesis is that this information alone is sufficient to discriminate between situations in which the robot operates as intended and situations in which the presence of one or more faults hampers its performance. In order to learn to detect the relevant changes in the flow of information caused by a fault, we simulate faults in the on-board software. We apply a well-establish technique known as *software implemented fault injection* (SWIFI) used in dependable systems research. The technique is usually applied to measure the robustness and fault tolerance of software systems (Hsueh et al., 1997, Arlat et al., 1990). In our case, we inject faults to discover how sensory readings and the control signals sent to the actuators by the control program change when the fault occurs. The idea is that by controlling the exact point in time, the location, and the type of the fault, and by recording the resulting flow of data in and out of the control program, we can use supervised learning techniques and obtain a classifier that, based on the data flow, can determine the state of the system.

We use time-delay neural networks (TDNNs) as classifiers (Waibel et al., 1989, Clouse et al., 1997). TDNNs are feed-forward networks that allow reasoning based on time-varying inputs without the use of recurrent connections. In a TDNN, the values for a group of neurons are assigned based on a set of observations from a fixed distance into the past. The TDNNs used in this study are normal multilayer perceptrons for which the inputs are taken from multiple, equally spaced points in a delay-line of past observations. TDNNs have been extensively used for time-series prediction due to their ability to make predictions based on data distributed in time. Unlike more elaborate, recurrent network architectures, the properties of multilayer TDNNs are well-understood and supervised learning through back-propagation can be applied.

3.1 Formal Definitions

Our aim is to obtain a function that maps from a set of current and past sensory inputs, S, and control signals, A, to either 0 or 1 corresponding to *no-fault* and *fault*, respectively:

$$g : S, A \rightarrow \{0, 1\}. \tag{1}$$

We assume that such a function exists and we approximate it with a feed-forward neural network. We let $I \subseteq (S \cup A)$ be the inputs to the network, and we choose a network that has a single output neuron and whose output is in the interval $(0, 1)$. The output is interpreted in a task-dependent way. For instance, a threshold-based classification scheme can be applied where an output value above a given threshold is interpreted as 1 (*fault*), whereas an output value below the threshold is interpreted as 0 (*no-fault*).

Sensory Inputs, Control Signals and Fault State: We perform a number of runs each consisting of a number of control cycles (sense-compute-act loops). For each control cycle, c, we record the sensory inputs and control signals to and from the control program. We let i^r_c denote a single set of control program inputs and outputs (CPIO), that is, the CPIO for control cycle c in run r. We let s denote the number of values in a single CPIO set. We let I^r be the ordered set of all CPIO sets for r. Similarly, for each control cycle we let f^r_c denote the fault state for control cycle c in run r, where $f^r_c = 1$ if a fault has been injected and 0 otherwise. Hence, $f^r_c = 0$ when the robot is operating normally and $f^r_c = 1$ otherwise.

Tapped Delay-Line and Input Group Distance: The CPIO sets are stored in a tapped delay-line, where each tap has a size s. The input layer of a TDNN is logically organized in a number of *input groups* $g_0, g_1, \ldots, g_{n-1}$ and each group consists of precisely s neurons, that is, one neuron for each value in a CPIO set. The activation of the input neurons in group g_t are then set according to $g_t = i^r_{c-t\cdot d}$, where c is the current control cycle and d is the *input group distance*. See Fig. 2 for an example. If we choose an input group distance $d = 1$, for example, the TDNN has access to the current and the $n - 1$ most recent CPIOs, whereas if $d = 2$, the TDNN has access to the current and every other CPIO set up to $2(n - 1)$ control cycles into the past, and so on. In this way, the input group distance specifies the relative distance in time between the individual groups and (along with the number of groups) how far into the past a TDNN 'sees'.

TDNN Structure and Activation Function: The input layer of the TDNN is fully connected to a hidden layer, which is again fully connected to the output layer. The output layer consists of a single neuron whose value reflects the network's classification of the current inputs. The activations of the neurons are computed layer-by-layer in a feed-forward manner and the following sigmoid activation function is used to compute the neurons' outputs in the hidden and the output layers:

$$f(a) = \frac{1}{1 + e^{-a}}, \tag{2}$$

where a is the activation of the neuron.

Classification and Learning: The output of the TDNN has a value between 0 and 1. The error factor used in the back-propagation algorithm is computed as the difference between the fault state f^r_c and the output o_c:

$$E_c = f^r_c - o_c. \tag{3}$$

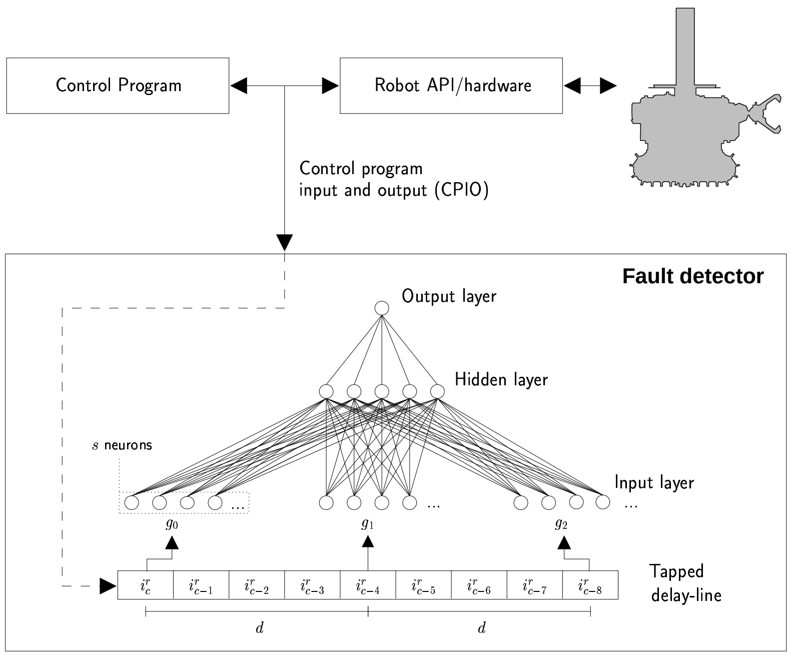


Fig. 2: An illustration of a fault detection module based on a TDNN. The current control program input and output (CPIO) is stored in the tapped delay-line and the activations of the neurons in the logical input groups are set according to the current and past CPIOs. In the example illustrated, there are 3 input groups and the input group distance d is 4.

The neural networks are all trained by a batch learning back-propagation algorithm that aims at minimizing the absolute value of the error factor E_c in (3).

In summary, sensor and actuator data is collected from a number of runs on real robots and different types of faults are injected. A TDNN is trained to discriminate between normal and faulty operation. By storing past observations in a tapped delay-line, classification based on how the flow of information changes over time is performed.

3.2 Robot Hardware

We use a number of real robots known as *s-bots* (Mondada et al., 2005). The *s-bot* platform has been used for several studies, mainly in swarm intelligence and collective robotics (Dorigo et al., 2004, Trianni and Dorigo, 2006, Nouyan and Dorigo, 2006). Overcoming steep hills and transport of heavy objects are notable examples of tasks which a single robot could not solve individually, but which has been solved successfully by teams of collaborating robots (Groß et al., 2006, O'Grady et al., 2005, Nouyan et al., 2006).

Each *s-bot* is equipped with an Xscale CPU running at 400 MHz, a number of sensors including an omni-directional camera, light and proximity sensor, as well as a number of actuators including a ring of 8x3 (RGB) colored leds, and a gripper that allows robots to attach to each other. The sensors and actuators are indicated in Fig. 3.

4 Fault Model

In this study, we focus on faults in the mechanical system that propels the *s-bots*. This system consists of a set of differential *treels* (combined tracks and wheels). Given that the treels contain

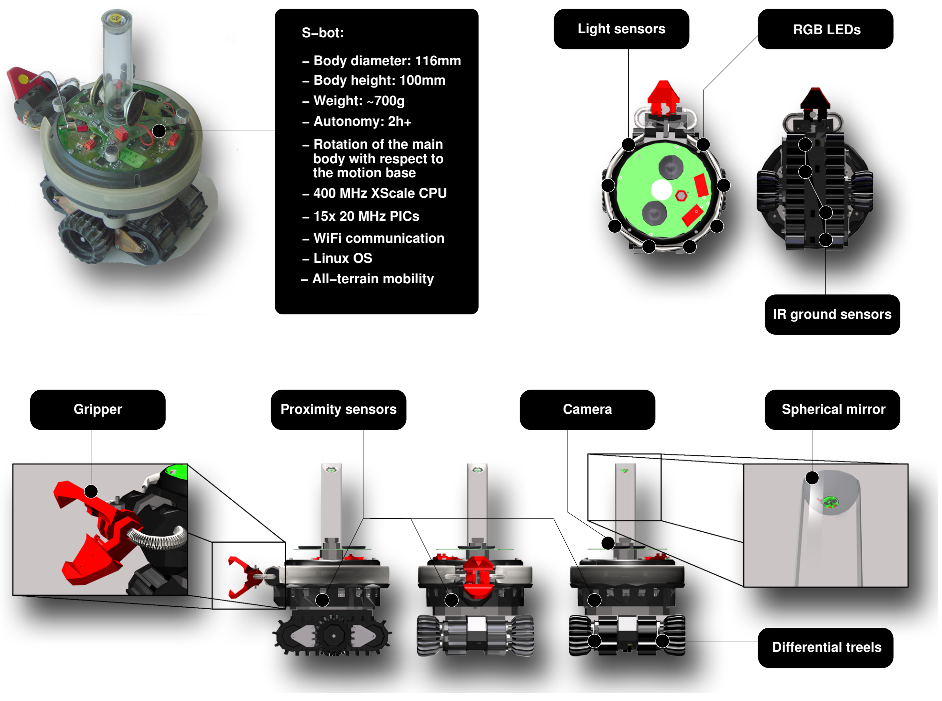


Fig. 3: The *s-bot* platform, sensors, and actuators.

moving parts and that they are used continuously in most experiments, they are the components in which the majority of faults occur. We analyze two types of faults. Both types can either be isolated to the left or the right treel or they can affect both treels simultaneously. The first type of fault causes one or both treels to stop moving. This usually happens if the strap that transfers power from the electrical motors to the treels breaks or jumps out of place. Whenever this happens, the treels stop moving. We denote this type of fault as *stuck-at-zero*.

The second type of fault occurs when an *s-bot's* software sub-system crashes leaving one (or both) motor(s) driving the treels running at some undefined speed. The result is that a treel no longer can be controlled by the on-board software. We refer to this type of fault as *stuck-at-constant*.

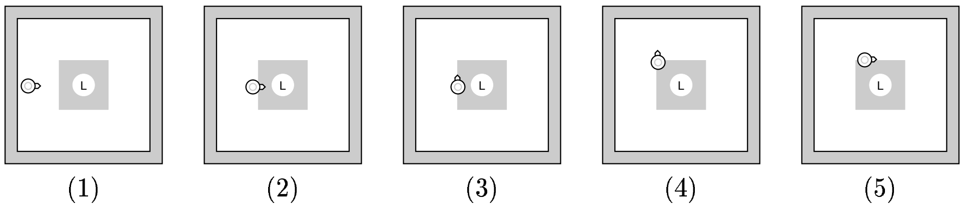
When data is collected, a number of runs are performed and in each run the *s-bot* starts in perfect condition. During the run, a fault is injected. The fault is simulated by ignoring the control program's commands to the failed part and by substituting them according to the type of fault injected. If, for instance, a *stuck-at-constant* fault is simulated in the left treel, the speed of that treel is set to a random value, and all future changes in speed requested by the control program are ignored.

5 The Three Tasks

We have chosen three tasks to study fault detection based on fault injection and learning. The tasks are called *find perimeter*, *follow the leader*, and *connect to s-bot*, respectively, and they are described in Fig. 5. For all tasks, we use a 180x180 cm arena surrounded by walls.

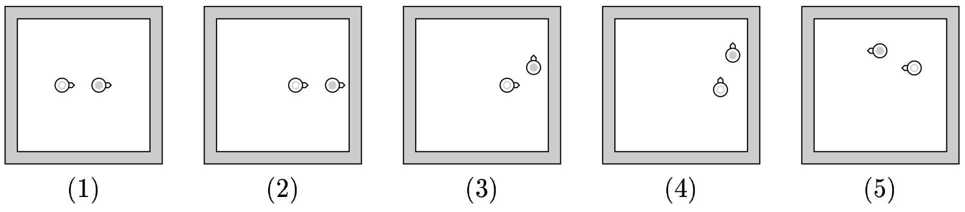
In the *find perimeter* task an *s-bot* follows the perimeter of a dark square drawn on the arena surface. In this task, the four infra-red ground sensors are used to discriminate between the normal light-colored arena surface and the dark square.

Find perimeter: An *s-bot* is situated in an arena with a square drawn with a dark color on an otherwise light floor. A light source is placed in the center of the square and the task for the *s-bot* is to follow the perimeter of the square.



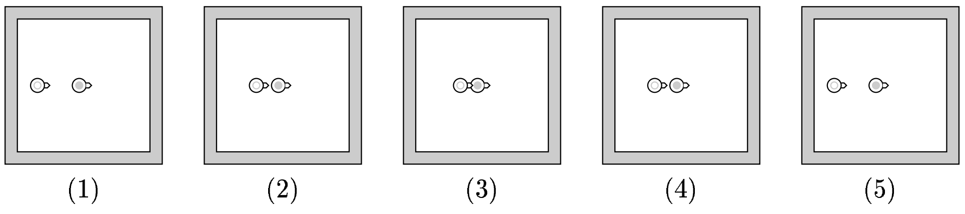
Sensors: IR ground (4 inputs), light (8 inputs)
Control period: 100 ms

Follow the leader: Two *s-bots* are placed in a square environment bounded by walls. One of the *s-bots* has been preassigned the *leader* role, while the other has been preassigned the *follower* role. The *leader* moves around in the environment. The *follower* tails the *leader* and tries to stay at a distance of 35 cm. If the *follower* falls behind, the *leader* waits. Faults are injected in the *follower* only.



Sensors: Camera (16 inputs), IR proximity (15 inputs)
Control period: 150 ms

Connect to robot: One *s-bot* attempts to physically connect to a stationary *s-bot* using its gripper. When a successful connection has been made, the *s-bot* waits for 10 seconds, disconnects, moves back, and tries to connect again. Faults are injected in the active *s-bot* only.



Sensors: Camera (16 inputs), optical sensors in gripper (4 inputs)
Control period: 150 ms

Fig. 4: Description of the three task: *find perimeter*, *follow the leader*, and *connect to s-bot*. For each task a list of sensors used and the control cycle period for the controllers are shown. The number in brackets after each sensor listed corresponds to the number of input values the sensor provides to the fault detector at each control cycle.

In the *follow the leader* task, an *s-bot* (the *leader*) performs a random walk in the environment and another *s-bot* (the *follower*) follows. The two robots perceive one another using the omni-directional camera. The infra-red proximity sensors are used to detect and avoid walls. Objects up to 50 cm away can be seen reliably through the camera. Infra-red proximity sensors have a range from a few centimeters up to 20 cm depending on the reflective properties of the obstacle or object. Faults are injected in the *follower* only.

In the *connect to s-bot* task, one *s-bot* tries to connect to another, stationary *s-bot*. The connection is made using the gripper. The connecting *s-bot* uses the camera to perceive the location of the other robot. Faults are only injected in the active *s-bot*.

Readings from sensors like an infra-red ground sensor are straight-forward to normalize and feed to the input neurons of a neural network. The camera sensor, in contrast, captures 640x480 color images. For these more complex sensor readings to serve as input to a neural network, relevant information must be extracted and processed beforehand. The *s-bots* have sufficient on-board processing power to scan the entire image and identify objects based on color information. The image processor is configured to detect the location of colored leds of the *s-bot's* only, and discard any other information. Since the robots operate on flat terrain and the images are reflections of a semi-spherical mirror, the distance in pixels from the center of the image corresponds to the physical distance between the robot and the perceived object. In order to make this information available to a neural network, we divide the image into 16 non-overlapping slices of equal size in terms of the field of view they cover. Each slice corresponds to a single input value. The value is set depending on distance to the closest object perceived in the slice. If no object is perceived, the value for a slice is 0. In this way, the camera sensor is effectively a range sensor for colored leds.

6 Experimental Setup

6.1 Data Collection

A total of 60 runs on real *s-bots* are performed for each of the three tasks. In each run, the robot(s) start in perfect condition, and at some point during the run a fault is injected. The fault is injected at a random point in time after the first 5 seconds of the run and before the last 5 seconds of the run according to a uniform distribution. There is a 50% probability that a fault affects both treels instead of only one of the treels, and faults of the type *stuck-at-zero* and *stuck-at-constant* are equally likely to occur. Each run consists of 1000 control cycles and for each control cycle the sensory inputs, control signals, and the current fault state are recorded. For the *find perimeter* task 1000 cycles correspond to 100 seconds, while for the *follow the leader* and the *connect to s-bot* tasks 1000 cycles correspond to 150 seconds, due to the longer control cycle period.

6.2 Training and Evaluation Data

The data sets for each task are partitioned into two subsets, one consisting of data from 40 runs, which is used for training; and one consisting of the data from the remaining 20 runs, which is used for a final performance evaluation. The TDNNs all have a hidden layer of 5 neurons and an input layer consisting of 10 input groups.

6.3 Performance Evaluation

The performance of the trained neural networks is computed based on the 20 runs reserved for evaluation for each task. A network is evaluated on data from one run at a time, and the output of the network is recorded and compared to the fault state.

Fault detection for autonomous robots involves reasoning based on partial and imperfect information due to limited and noisy sensors and actuators. Furthermore, there is often a delay between the occurrence of a fault and observable symptoms. Since the classification is binary (is there a fault or not?), the scheme for approximating the fault state must be chosen according to

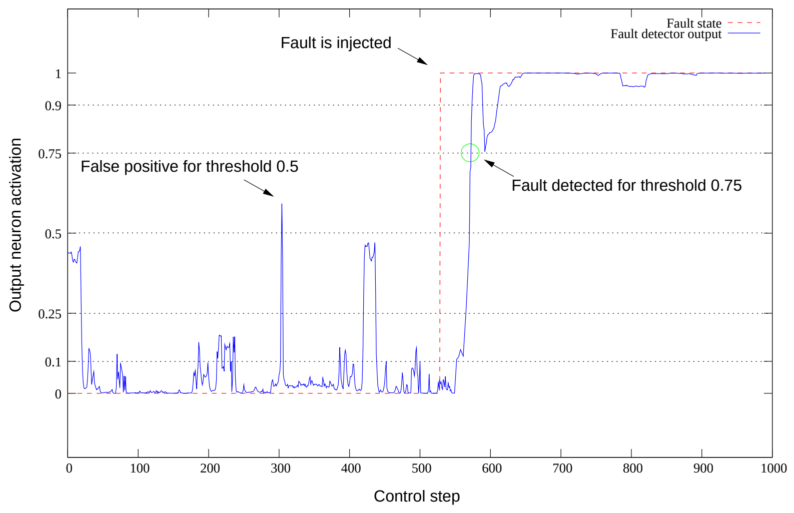


Fig. 5: An example of the output of a trained TDNN during a run. The dotted line shows the optimal output. At control cycle 529 a fault is injected. Five different thresholds are indicated, 0.10, 0.25, 0.50, 0.75, and 0.90, and a false positive for threshold 0.50 is shown at control cycle 304 (the output has a value greater than 0.50 *before* the fault was injected at control cycle 529). The latency for a threshold is the number of control cycles from the moment the fault is injected till the moment the output value of the TDNN becomes greater than the threshold. In the example above, the latency for threshold 0.75 is 43 control cycles because the output of the TDNN reaches 0.75 only at control cycle 562, that is, 43 control cycles after the fault was injected.

the desired properties of the fault detector. For instance, the output of a trained TDNN could be interpreted such that any value above a given threshold triggers the fault recovery mechanism. We present results for five different thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90.

A graphical representation of an evaluation run is shown in Fig. 5. In the run shown, a fault was injected at control cycle 529. Choosing a threshold of 0.50 would result in a false positive at control cycle 304, since the output of the network is higher than 0.50 *before* a fault has been injected. If we choose a higher threshold, either 0.75 or 0.90, false positives are avoided. However, choosing a higher threshold has a negative impact on another aspect of a fault detector's performance, namely its *latency*. The latency is the number of cycles between fault injection and fault detection. In the example in Fig. 5 the fault is detected at control cycle 553, 561, 570, 572, and 574 for the thresholds 0.10, 0.25, 0.50, 0.75, and 0.90, yielding latencies of 24, 32, 41, 43, 45 control cycles, respectively.[1]

For some tasks, the recovery procedure is costly, and fewer false positives might be desirable even at the cost of a higher latency. For other tasks undetected fault can have serious consequences and a low latency is more important than reducing the number of false positives. The trade-off between latency and false positives can be controlled by the way in which we choose to interpret the output of the fault detector, and a simple way of doing so is by choosing a threshold.

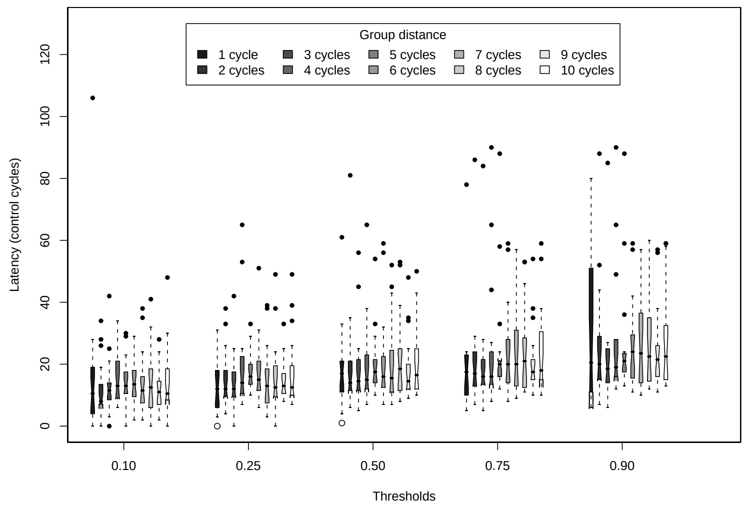


Fig. 6: Latency results for different input group distances for the *find perimeter* task.

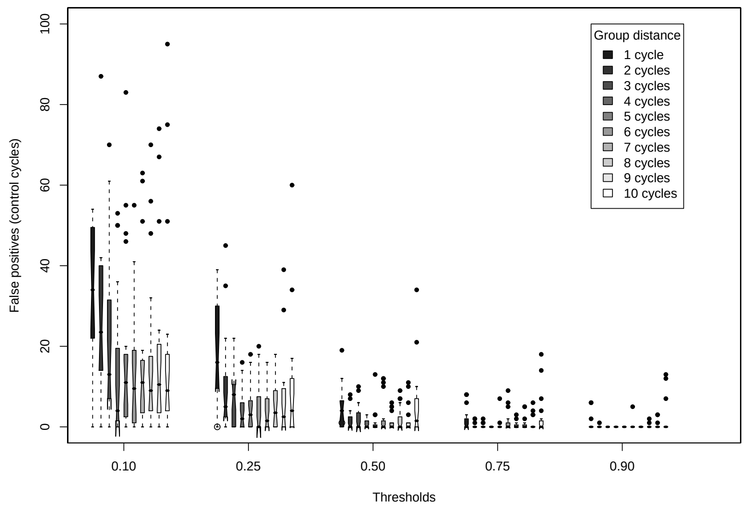


Fig. 7: False positive results for different input group distances for the *find perimeter* task.

7 Results

Fig. 6 and Fig. 7 show respectively the latency and the false positive results for the *find perimeter* task for input group distances ranging from 1 to 10 control cycles. Results are shown for the five

[1] It is important to note that a latency of 24 control cycles may seem long, but the faults that we are trying to detect do not necessarily have an immediate impact on the performance of a robot. If, for instance, the fault injected is a blocked treel (a fault of the type *stuck-at-zero*), the fault can only be detected if the control program tries to set the treel to a non-zero speed. Furthermore, if the control program only sets low speeds (values close to zero) it might take long before a fault can be detected.

Table 1: Number of undetected faults (out of a maximum of 20) for the *find perimeter* task, for different thresholds and input group distances *d*.

		Threshold				
		0.10	0.25	0.50	0.75	0.90
Input group distance	1	0	1	1	2	2
	2	1	1	1	3	3
	3	0	0	0	1	3
	4	0	0	1	1	2
	5	0	0	0	1	1
	6	0	0	0	0	0
	7	0	0	0	0	0
	8	0	0	0	0	0
	9	0	0	0	0	0
	10	0	0	0	0	0

chosen thresholds. The general trend is that latency is not significantly affected by the chosen input group distances, whereas the number of false positives is comparatively high for low input group distances.

An input groups distance of 1 means that the TDNN is provided data from the past 10 control cycles (because there are 10 input groups). 10 control cycles are equal to 1 second in the *find perimeter* task. Similarly, an input group distance of 2 means that the TDNN is provided data from the past 2 seconds, but only from every other control cycle. The false positive results indicate that data from a period longer than 2 seconds is needed for a more accurate classification.

In some cases, a fault is never detected. This can either be because a fault is injected at the end of an experiment and the fault detector does not have a sufficient amount of time to detect it, or because the behavior after fault injection closely resembles correct behavior. The number of undetected faults for different thresholds and input group distances is shown in Tab. 1. All undetected faults were observed when low input group distances were used.

The false positive and the latency results for the *follow the leader* and *connect to s-bot* tasks are shown in Fig. 8 and Fig. 9, respectively. In both cases, an input group distance of 5 was used. The number of undetected faults for the two tasks is shown in Tab. 2. Two interesting tendencies should be noticed: The number of false positives for the *follow the leader* task are comparatively high, while for the *connect to s-bot* the latencies are high when compared to the results for the other tasks. In the *follow the leader* task there are two robots moving around and the fault detector for the *follower*, in which faults were injected, has to infer the presence of faults based on the interaction between itself and the *leader*. The misclassification of the current state can occur in situations where, for instance, the *leader* and the *follower* are moving at constant speeds in the same direction. In those cases the *follower* will essentially receive sensory inputs similar to those in situation where both its treels are *stuck-at-zero*: The *leader* waits for the *follower*, but due to the fault, the *follower* does not move. The fact that the control program (and therefore the fault detector) depends on a dynamic feature of the environment (the *leader*) seems to complicate accurate classification of the fault state. However, fault detection is still quite good, especially considering that the *leader* often is the *only* object perceivable by the *follower* unless it is close enough to a wall for the proximity sensors to detect it.

The comparative high latency results for the *connect to s-bot* task are likewise due to a task-dependent feature: After a successful connection has been made, the connecting robot waits for 10 seconds before disconnecting, moving back, and attempting to make a new connection. During the waiting period it is not possible to detect if a fault has occurred in the treels or not. Even if a *stuck-at-constant* fault is injected, causing one or both treels to be assigned a random and non-changeable speed, the outcome is the same: The robot does not move because it is physically connected to the other robot. Thus, it can take longer to detect a fault given the existence of these particular situations in which a fault does not have an effect on the performance of the robot.

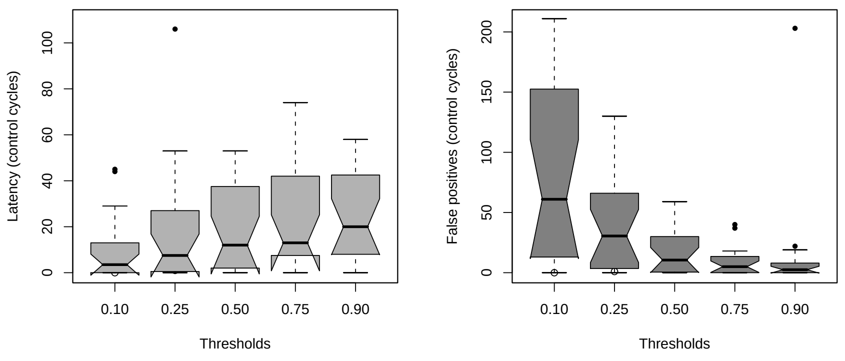


Fig. 8: Results for the *follow the leader* task, for different thresholds and an input group distance of 5.

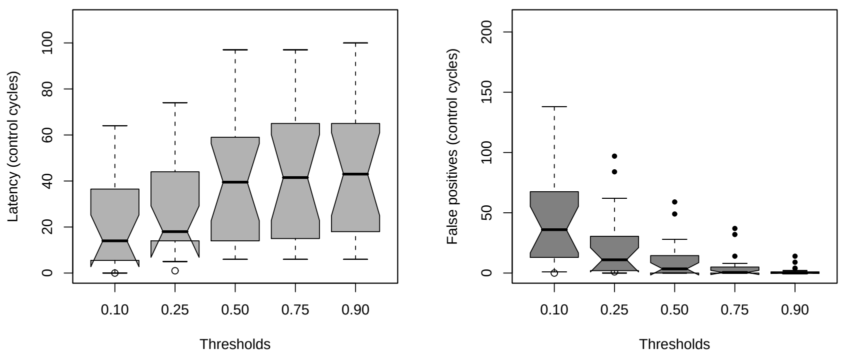


Fig. 9: Results for the *connect to s-bot* task, for different thresholds and an input group distance of 5.

We experimented with a different interpretation mechanism for the output of the fault detecting neural network. Given that the majority of the false positives occur for a single or few consecutive control cycles only (like in the example in Fig. 5), we have tried to reduce the number of false positives by filtering out spikes. This is done by computing a moving average of the output value of the fault detector and setting a threshold of 0.75. The corresponding results for latencies and false positives are shown in Fig. 10 and Fig. 11, respectively, using a moving average over 25 control cycles.

By computing the moving average and smoothing the output of the TDNN, we almost completely eliminate false positives. As the results in Fig. 10 show, however, this is at the cost of a higher latency. Since the moving average increases latency, it can result in more undetected

Table 2: Number of undetected faults (out of a maximum of 20) for the *follow the leader* and *connect to s-bot* tasks, for different thresholds and an input group distance of 5.

	Threshold				
	0.10	0.25	0.50	0.75	0.90
Follow the leader	0	0	0	0	0
Connect to s-bot	1	2	2	2	3

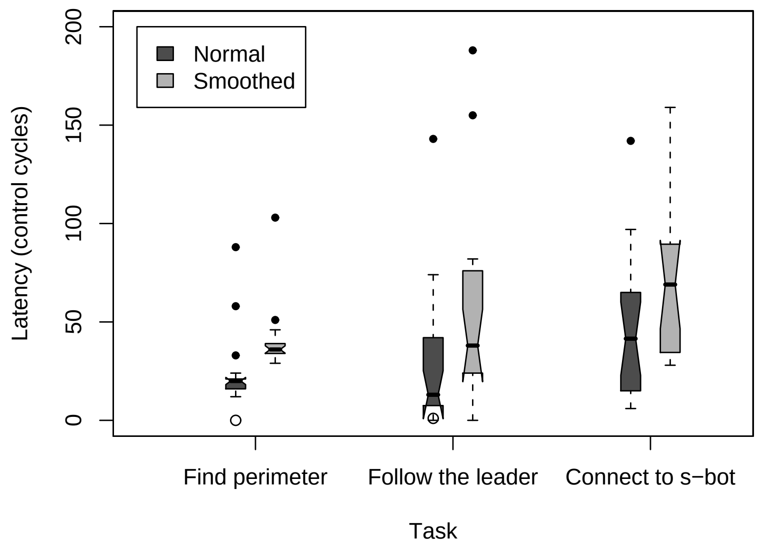


Fig. 10: Latency results for a fault detector in which the output of the TDNNs is used directly and one in which the output is smoothed by computing the moving average over 25 control cycles.

faults as more runs finish before faults are detected. For the *find perimeter* task, 2 faults were not detected when averaging the output over 25 control cycles, compared to only 1 when averaging was not used. Similarly, for the *connect to s-bot* task 5 faults were not detected when a moving average was used, compared to 2 when the output of the TDNN was used directly. For the *follow the leader* task all faults were detected in both cases.

8 Discussion

The results presented in this study suggest that fault detection through fault injection and learning is a viable method for obtaining a fault detector for autonomous mobile robots. All results are based on experiments with real robots. The robots were not equipped with dedicated or redundant sensors to improve fault detection. In order to detect faults, only the information flowing between the control program and the robots' sensors and actuators was used.

Although the latency and the number of false positives detected could likely be reduced using more data from (possibly dedicated) sensors, the results show that a fairly small amount of key information is sufficient to obtain a good fault detector. It was shown that when we averaged the

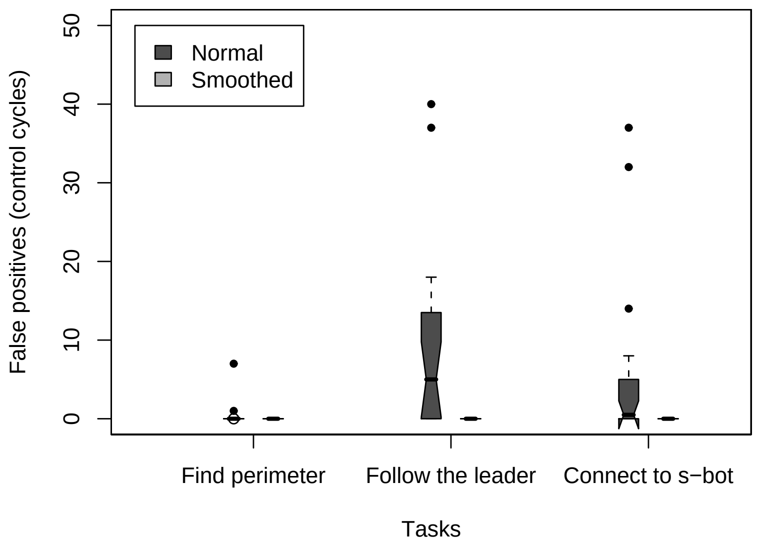


Fig. 11: False positives results for a fault detector in which the output of the TDNNs is used directly and one in which the output is smoothed by computing the moving average over 25 control cycles. Only one run for the *follow the leader* task resulted in false positives when the TDNN output was smoothed. The run is not shown in the figure since it is out of scale (164 false positives were detected during this run).

output of the fault detectors over several control cycles, we obtained a correct detection rate of 75% - 100%, without any false positives except in a single case.

Our fault detector distinguished between normal and faulty behavior based on what it learned during normal operation and after faults occurred. An obvious extension would be to include fault diagnosis, that is, not only to detect the presence of the fault, but also its location. This could be useful when, for instance, a gripper breaks during transport of a heavy object. If the control program is made aware of this fault, it could steer the robot to push the object instead of unsuccessfully trying to connect to and pull the object. One way of extending the proposed methodology to include fault identification is to add more output neurons to the classifying neural network. Different output neurons would then correspond to different types of faults. Another approach is to use multiple neural networks, one for each fault type. In addition, in this study we have focused on mechanical faults, since they frequently occur in mobile robots. A future activity will be to determine how well other types of faults, such as faults in sensors and other actuators, can be detected.

In contrast with the majority of existing approaches to fault detection, the method proposed in this study is model-free. An interesting consequence of relying on our model-free approach is that the scheme is straightforward to extend beyond self-diagnosis. For instance, in the *follow the leader* task, we injected faults in the *follower* robot and a fault detector for that robot was trained. However, a fault occurring in the *follower* has, in most cases, influence on what the *leader* robot perceives and its subsequent actions. A fault detector in the *leader* should, therefore, be able to detect faults in the *follower*. We are currently verifying whether this is the case and studying methods for scaling the proposed approach to whole colonies of robots.

Acknowledgements

Anders Christensen and Rehan O'Grady acknowledge support from COMP2SYS, a Marie Curie Early Stage Research Training Site funded by the European Community's Sixth Framework Programme (grant MEST-CT-2004-505079). The information provided is the sole responsibility of the authors and does not reflect the European Commission's opinion. The European Commission is not responsible for any use that might be made of data appearing in this publication. This work was supported by the ANTS project, an *Action de Recherche Concertée* funded by the Scientific Research Directorate of the French Community of Belgium. Marco Dorigo acknowledges support from the Belgian FNRS, of which he is a Research Director.

References

Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J., Laprie, J., Martins, E. and Powell, D. (1990), 'Fault injection for dependability validation: A methodology and some applications', *IEEE Transactions on Software Engineering* **16**(2), 166–182.

Canham, R., Jackson, A. and Tyrrell, A. (2003), Robot error detection using an artificial immune system, *in* 'Proceedings of NASA/DoD Conference on Evolvable Hardware, 2003', IEEE Computer Society, Washington D.C., pp. 199–207.

Carlson, J. and Murphy, R. (2003), Reliability analysis of mobile robots, *in* 'Proceedings of IEEE International Conference on Robotics and Automation, ICRA'03', Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 274–281.

Clouse, D., Giles, C., Horne, B. and Cottrell, G. (1997), 'Time-delay neural networks: Representation and induction of finite-state machines', *IEEE Transactions on Neural Networks* **8**, 1065–1070.

Dearden, R., Hutter, F., Simmons, R., Thrun, S., Verma, V. and Willeke, T. (2004), Real-time fault detection and situational awareness for rovers: Report on the Mars technology program task, *in* 'Proceedings of IEEE Aerospace Conference', Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, pp. 826–840.

Dorigo, M., Trianni, V., Şahin, E., Groß, R., Labella, T. H., Baldassarre, G., Nolfi, S., Deneubourg, J.-L., Mondada, F., Floreano, D. and Gambardella, L. M. (2004), 'Evolving self-organizing behaviors for a swarm-bot', *Autonomous Robots* **17**(2–3), 223–245.

Forrest, S., Perelson, A., Allen, L. and Cherukuri, R. (1994), Self-nonself discrimination in a computer, *in* 'Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy', Vol. 212, IEEE Computer Society Press, Los Alamitos, CA, pp. 202–212.

Gertler, J. J. (1988), 'Survey of model-based failure detection and isolation in complex plants', *IEEE Control Systems Magazine* **8**, 3–11.

Goel, P., Dedeoglu, G., Roumeliotis, S. and Sukhatme, G. (2000), Fault detection and identification in a mobile robot using multiple model estimation and neural network, *in* 'Proceedings of IEEE International Conference on Robotics and Automation, ICRA'00', Vol. 3, IEEE Computer Society Press, Los Alamitos, CA, pp. 2302–2309.

Groß, R., Bonani, M., Mondada, F. and Dorigo, M. (2006), 'Autonomous self-assembly in swarm-bots', *IEEE Transactions on Robotics* **22**(6), 1115–1130.

Hinchey, M., Rash, J., Rouff, C. and Truszkowski, W. (2004), 'NASA's swarm missions: the challenge of building autonomous software', *IT Professional* **6**, 47–52.

Hsueh, M., Tsai, T. and Iyer, R. (1997), 'Fault injection techniques and tools', *Computer* **30**(4), 75–82.

Isermann, R. (1997), 'Supervision, fault-detection and fault-diagnosis methods – An introduction', *Control Engineering Practice* **5**(5), 639–652.

Isermann, R. and Ballé, P. (1997), 'Trends in the application of model-based fault detection and diagnosis of technical process', *Control Engineering Practice* **5**(5), 709–719.

Kalman, R. (1960), 'A new approach to linear filtering and prediction problems', *Journal of Basic Engineering* **82**(1), 35–45.

Kochan, A. (2005), 'A bumper year for robots', *Industrial Robot: An International Journal* **32**, 201–204.

Lerner, U., Parr, R., Koller, D. and Biswas, G. (2000), Bayesian fault detection and diagnosis in dynamic systems, *in* 'Proceedings of the 7th National Conference on Artificial Intelligence', AAAI Press/The MIT Press, Cambridge, MA, pp. 531–537.

Li, P. and Kadirkamanathan, V. (2001), 'Particle filtering based likelihood ratio approach to fault diagnosis in nonlinear stochastic systems', *IEEE Transactions on Systems, Man and Cybernetics, Part C* **31**(3), 337–343.

Mondada, F., Gambardella, L. M., Floreano, D., Nolfi, S., Deneubourg, J.-L. and Dorigo, M. (2005), 'The cooperation of swarm-bots: Physical interactions in collective robotics', *IEEE Robotics & Automation Magazine* **12**(2), 21–28.

Nouyan, S. and Dorigo, M. (2006), Chain based path formation in swarms of robots, *in* M. Dorigo, L. M. Gambardella, M. Birattari, A. Martinoli, R. Poli and T. Stützle, eds, 'Ant Colony Optimization and Swarm Intelligence: 5th International Workshop, ANTS 2006', Vol. 4150 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin, Germany, pp. 120–131.

Nouyan, S., Groß, R., Bonani, M., Mondada, F. and Dorigo, M. (2006), Group transport along a robot chain in a self-organised robot colony, *in* T. Arai, R. Pfeifer, T. Balch and H. Yokoi, eds, 'Intelligent Autonomous Systems 9, IAS 9', IOS Press, Amsterdam, The Netherlands, pp. 433–442.

O'Grady, R., Groß, R., Mondada, F., Bonani, M. and Dorigo, M. (2005), Self-assembly on demand in a group of physical autonomous mobile robots navigating rough terrain, *in* M. S. Capcarrere, A. A. Freitas, P. J. Bentley, C. G. Johnson and J. Timmis, eds, 'Advances in Artificial Life: 8th European Conference, ECAL 2005', Vol. 3630 of *Lecture Notes in Artificial Intelligence*, Springer Verlag, Berlin, Germany, pp. 272–281.

Patton, R., Uppal, F. and Lopez-Toribio, C. (2000), Soft computing approaches to fault diagnosis for dynamic systems: A survey, *in* A. Edelmayer and C. Banyasz, eds, 'Proceedings of 4th IFAC Symposium on Fault Detection supervision and Safety for Technical Processes', Vol. 1, Elsevier, Oxford, UK, pp. 298–311.

Roumeliotis, S., Sukhatme, G. and Bekey, G. (1998), Sensor fault detection and identification in a mobile robot, *in* 'Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems', Vol. 3, IEEE Computer Society Press, Los Alamitos, CA, pp. 1383–1388.

Terra, M. and Tinos, R. (2001), 'Fault detection and isolation in robotic manipulators via neural networks: A comparison among three architectures for residual analysis', *Journal of Robotic Systems* **18**(7), 357–374.

Trianni, V. and Dorigo, M. (2006), 'Self-organisation and communication in groups of simulated and physical robots', *Biological Cybernetics* **95**, 213–231.

Vemuri, A. and Polycarpou, M. (1997), 'Neural-network-based robust fault diagnosis in robotic systems', *IEEE Transactions on Neural Networks* **8**(6), 1410–1420.

Verma, V., Gordon, G., Simmons, R. and Thrun, S. (2004), 'Real-time fault diagnosis', *Robotics and Automation Magazine, IEEE* **11**(2), 56–66.

Verma, V. and Simmons, R. (2006), 'Scalable robot fault detection and identification', *Robotics and Autonomous Systems* **54**(2), 184–191.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K. and Lang, K. (1989), 'Phoneme recognition using time-delay neural networks', *IEEE Transactions on Acoustics, Speech, and Signal Processing* **37**, 328–339.