Automatic Synthesis of Fault Detection Modules for Mobile Robots*

Anders Lyhne Christensen Rehan O'Grady Mauro Birattari Marco Dorigo
IRIDIA-CoDE, Université Libre de Bruxelles
50, Avenue Franklin Roosevelt, CP 194/6
1050 Brussels, Belgium
alyhne@iridia.ulb.ac.be,{rogrady,mbiro,mdorigo}@ulb.ac.be

Abstract

In this paper, we present a new approach for automatic synthesis of fault detection modules for autonomous mobile robots. The method relies on the fact that hardware faults typically change the flow of sensory perceptions received by the robot and the subsequent behavior of the control program. We collect data from three experiments with real robots. In each experiment, we record all sensory inputs from the robots while they are operating normally and after software-simulated faults have been injected. We use back-propagation neural networks to synthesize task-dependent fault detection modules. The performance of the modules is evaluated in terms of false positives and latency.

1 Introduction

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 [12]. The problem is often exacerbated if the fault is not detected in a timely manner. In a recent paper [3], the reliability of seven mobile robots from three different manufacturers was tracked over a period of two years and the mean time between failures was found to be 8 hours. The result suggests that faults in mobile robots are quite frequent. As more and more robots are introduced in space, industry, and private homes, fault detection is becoming an increasingly important issue to address. Fault detection can be achieved by adding special-purpose hardware such as torque and joint position sensors [23]. Adding additional hardware increases cost and complexity, and it is therefore something that we would like to avoid in many cases.

In this paper, we propose a general method for synthesizing fault detection modules for autonomous robots. The method is not computationally intensive and requires no special-purpose hardware. In brief, sensory data and control signals to actuators are recorded, 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 faults have been injected. Neural networks are trained on the recorded data to infer the presence of faults from changes in the flow of sensory inputs and control program behavior.

2 Related Work

Fault detection is based on observations of a system's behavior. Deviations from normal behavior can be interpreted as symptoms of a fault in the system. A large body of research in *model-based fault detection* approaches exists [8, 13]. 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 [25, 23, 21].

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. Such a fault detection system determines that a fault corresponding to a particular model is present 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 [22, 9]. Kalman filters are based on the assumption that the modelled system can be approximated as a Markov chain built on linear op-

* An extended version of this paper has been submitted for journal publication.

erators perturbed by Gaussian noise [14]. 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 [15]. Recently, computationally efficient approaches for approximating Bayesian belief using *particle filters* have been studied as a means for fault detection and identification [5, 26, 16]. 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 [27].

Artificial immune-systems (AIS) represent a biologically inspired approach to fault detection. An AIS is a classifier that distinguishes between *self* and *non-self* [7]. 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 [2] 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

Our approach relies on training detectors on observation from normal operation and after a fault has been injected (as opposed to AIS in which training of detectors is only performed on observation from normal operation). Furthermore, we base fault detection on both current and past observations. 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 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, e.g., if 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 sensors.

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 affecting the robot's behavior. 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. The approach involves simulating hardware faults in the on-board software. We apply a well-establish technique known as *software implemented fault injection* (SWIFI), which is mainly used in dependable systems research. The technique is usually applied to measure the robustness and fault tolerance of software systems [11, 1]. In our case, we inject faults to discover how both the sensor readings and the control signals to the actuators change when faults are present. In this way, we can control the exact point in time at which the fault occurs, the location, and the type of the fault. By recording the resulting flow of data in and out of the control program, we can use supervised learning techniques and synthesize a classifier that, based on the data flow, can determine if the system is in a fault state or not.

Each fault detection module consists of a time-delay neural network (TDNN) [28, 4]. TDNNs are feed-forward networks that allow for classification 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 paper are standard 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 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:

$$g : S, A \rightarrow \{0, 1\}. \quad (1)$$

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*). In the following, we describe our approach in more detail:

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_c^r 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_c^r denote the fault state for control cycle c in run r, where $f_c^r = 1$ if a fault has been injected and 0 otherwise. Hence, $f_c^r = 0$ when the robot is operating normally and $f_c^r = 1$ 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 $g_t = i_{c-t \cdot d}^r$, where c is the current control cycle and d is the *input group distance*. See Fig. 1 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_c^r and the output o_c:

$$E_c = f_c^r - o_c. \tag{3}$$

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).

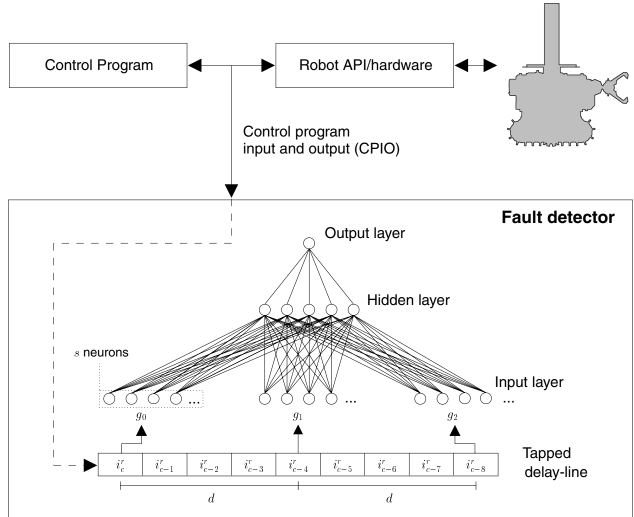


Figure 1: An illustration of a fault detection module based on a TDNN with 3 input groups and an input group distance (d) of 4.

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 can be performed.

3.2 Robot Hardware

We use a number of real robots known as *s-bots* [17]. The *s-bot* platform has been used for several studies, mainly in swarm intelligence and collective robotics [6, 24, 18]. Overcoming steep hills and transport of heavy objects are notable examples of tasks that a single robot could not solve individually, but that have been solved successfully by teams of collaborating *s-bots* [10, 20, 19].

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. 2.

4 Fault Model

In this paper, 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) [17]. Given

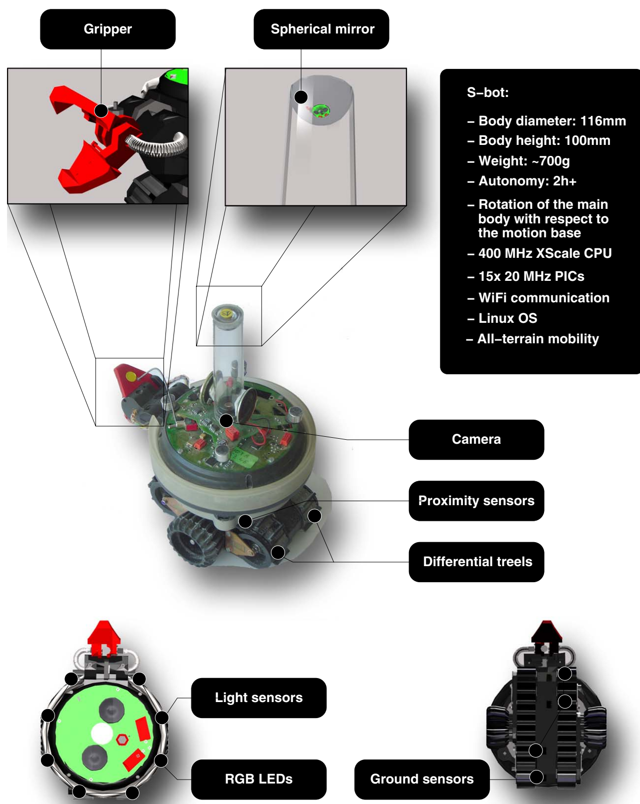
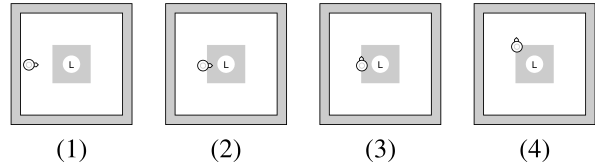
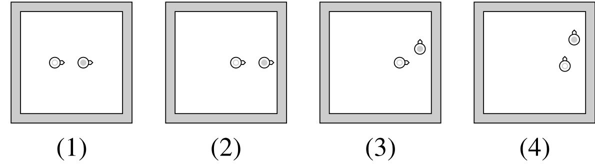


Figure 2: The *s-bot* platform, sensors, and actuators.

Find perimeter:



Follow the leader:



Connect to s-bot:

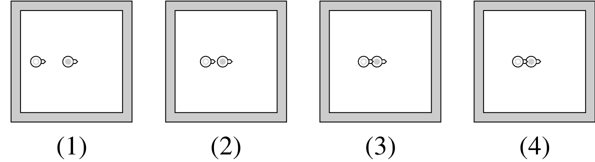


Figure 3: The three task.

that the treels contain 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. 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. The consequence of a fault depends on the type and location of the fault and on the subsequent actions performed by the control program. The control program is unaware that a fault has occurred and continues to try to steer the robot based on the sensory input it receives.

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. They are illustrated in Fig. 3. 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. The four infrared ground sensors are used to discriminate between the normal light-colored arena surface and the dark square. A light source detectable by the light sensors is placed in the center of the square.

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 infrared proximity sensors are used to detect and avoid walls. Depending on light conditions, objects up to 50 cm away can be seen reliably through the camera. Infrared 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 connecting *s-bot*.

In the *find perimeter* task, each control cycle period is 100 ms, while for the *follow the leader* and the *connect to s-bot* tasks the control cycle period is 150 ms. For the latter two tasks, a longer control cycle period is required in order to have time to extract the relevant information from the images captured by the camera. Images from the camera are partitioned into 16 sections and each of the 16 neurons are assigned values inversely proportional to the distance of the closest object perceived in the corresponding section. The image processor has been configured to detect colored LEDs. This configuration means that the camera detects other *s-bots* only and not objects like walls. For the other sensors such as the infrared ground sensors, readings are normalized before they are used as inputs to the TDNN in a fault detector.

6 Experimental Setup

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 final 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.

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 performance evaluation. The TDNNs all have a hidden layer of 5 neurons and an input layer consisting of 10 input groups.

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. The output of the network is recorded and compared to the fault state.

Fault detection for autonomous robots involves classification 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 the desired properties of the fault detector.

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 faults 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 our interpretation of the TDNN's output, and a simple way of doing so is by fine-tuning a threshold. For instance, the output of a trained TDNN could be interpreted such that any value above a chosen threshold triggers the fault recovery mechanism. In the next section, we present results for five different thresholds: 0.10, 0.25, 0.50, 0.75, and 0.90.

7 Results

The false positive and the latency results for the *find perimeter*, *follow the leader*, and *connect to s-bot* tasks are shown in Fig. 4, Fig. 5, and Fig. 6, respectively. For all tasks, an input group distance of 5 is used.

The fault detector synthesized for the *find perimeter* task in general detects faults with a lower latency and produces fewer false positives than the fault detectors for the other two tasks. We believe that this is due to both the set of sensors used and to the nature of the *find perimeter* task. In this task, the *s-bot* follows the perimeter of the dark square on the ground in a zig-zag motion, meaning that its ground sensors constantly detect the color of the underlying surface. Moreover, the light source placed in the center of the dark square provides a fixed point of reference to which the zig-zag motion can be correlated. We believe that the constant change in sensory input according to a regular pattern and a fixed point of reference are features that a fault detection module can exploit. In the *connect to s-bot* task, the stationary *s-bot* does provide a fixed reference, however, it is perceived through the camera by the other robot. The camera can be quite noisy, especially when a robot is moving. Furthermore, the connecting *s-bot* does not move according to a regular pattern with constant changes in sensory input. In the *follow the leader* task, no fixed point of reference exists.

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. In the *follow the leader* task there are two robots moving around. 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 mov-

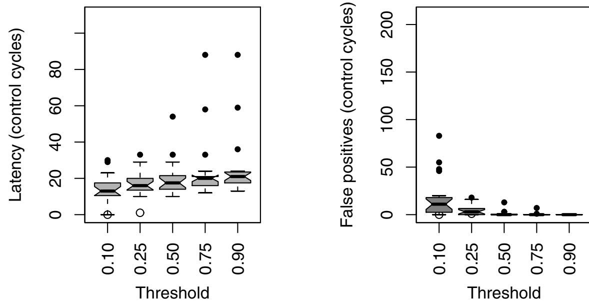


Figure 4: Results for the *find perimeter* task, for different thresholds and an input group distance of 5.

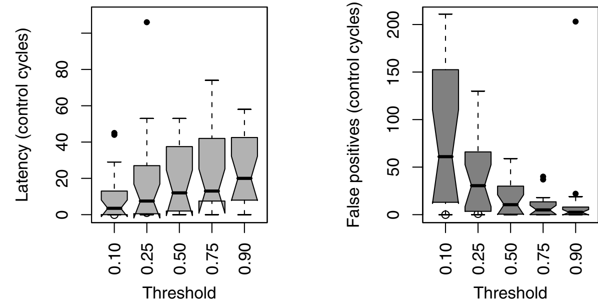


Figure 5: Results for the *follow the leader* task, for different thresholds and an input group distance of 5.

ing 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*, i.e., in the case where 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 within proximity sensor range of a wall.

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.

We experimented with a different interpretation mechanism for the output of the fault detecting neural network. The majority of the false positives occur only for a single control cycle or for a few consecutive control cycles. We tried to reduce the number of false positives by filtering out these spikes: we computed the moving average of the output value of the fault detector and used a threshold of 0.75. The corresponding results for latencies and false positives are shown in Fig. 7, using a moving average over 25 control

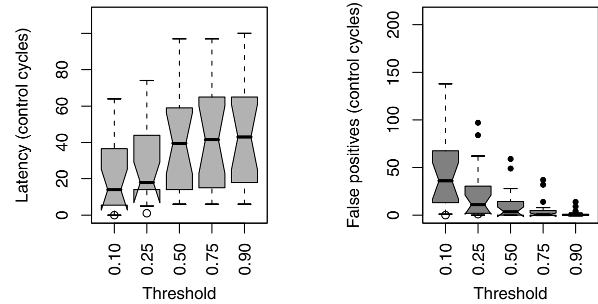


Figure 6: Results for the *connect to s-bot* task, for different thresholds and an input group distance of 5.

cycles.

By computing the moving average and smoothing the output of the TDNN, we almost completely eliminate false positives (during one *follow the leader* run the fault detector produced 164 false positives). As the results in Fig. 7 show, however, this is at the cost of a higher latency.

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. For the *find perimeter* task, 2 out of 20 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

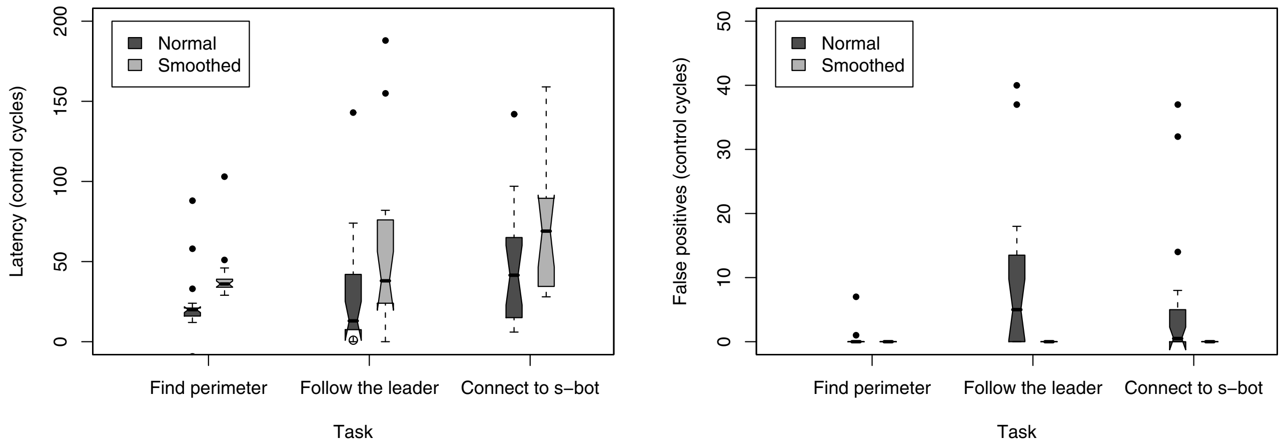


Figure 7: Latency and false positives results for a fault detector in which the output of the TDNNs is used directly and one in which the output is smoothed by computing the moving average over 25 control cycles.

detected in both cases.

8 Conclusions

The results presented in this paper suggest that fault detection through fault injection and learning is a viable method for synthesizing fault detectors 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.

It was shown that when we averaged the 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). The results show that a fairly small amount of key information is sufficient to obtain good fault detectors. The latency and the number of false positives could probably be reduced using data from more (possibly dedicated) sensors. Additional sensors would increase the complexity of the hardware and the system designer is therefore faced with a trade-off between cost and accuracy. The method we proposed is, however, still equally applicable if additional and/or dedicated fault detection sensors are used.

Our fault detector learned to distinguish faulty behavior from normal behavior based on observations from experiments in which faults were simulated. An obvious extension would be to include fault diagnosis, that is, not only to detect the presence of a 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 the object and pull it. One way of extending our methodology to include fault identification is to add more output neurons to the neural network. Different output neurons would then correspond to different types of faults. Another approach would be to use multiple neural networks, one for each fault type.

In contrast with the majority of existing approaches to fault detection, the method proposed in this study is model-free. An interesting property of our 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 exogenous fault detection.

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

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

[2] R. Canham, A. Jackson, and A. Tyrrell. Robot error detection using an artificial immune system. In *Proc. of NASA/DoD Conf. on Evolvable Hardware, 2003*, pages 199–207. IEEE Press, Washington D.C., 2003.

[3] J. Carlson and R. Murphy. Reliability analysis of mobile robots. In *Proc. of IEEE Int. Conf. on Robotics and Automation, ICRA'03*, volume 1, pages 274–281. IEEE Press, Piscataway, NJ, 2003.

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

[5] R. Dearden, F. Hutter, R. Simmons, S. Thrun, V. Verma, and T. Willeke. Real-time fault detection and situational awareness for rovers: Report on the Mars technology program task. In *Proc. of IEEE Aerospace Conf.*, pages 826–840. IEEE Press, Piscataway, NJ, 2004.

[6] M. Dorigo, V. Trianni, E. Şahin, R. Groß, T. Labella, G. Baldassarre, S. Nolfi, J.-L. Deneubourg, F. Mondada, D. Floreano, and L. M. Gambardella. Evolving self-organizing behaviors for a swarm-bot. *Autonomous Robots*, 17(2–3):223–245, 2004.

[7] S. Forrest, A. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *Proc. of the 1994 IEEE Symp. on Research in Security and Privacy*, pages 202–212. IEEE Press, Piscataway, NJ, 1994.

[8] J. Gertler. Survey of model-based failure detection and isolation in complex plants. *IEEE Control Systems Mag.*, 8:3–11, 1988.

[9] P. Goel, G. Dedeoglu, S. Roumeliotis, and G. Sukhatme. Fault detection and identification in a mobile robot using multiple model estimation and neural network. In *Proc. of IEEE Int. Conf. on Robotics and Automation, ICRA'00*, pages 2302–2309. IEEE Press, Piscataway, NJ, 2000.

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

[11] M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

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

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

[14] R. Kalman. A new approach to linear filtering and prediction problems. *Jour. of Basic Eng.*, 82(1):35–45, 1960.

[15] U. Lerner, R. Parr, D. Koller, and G. Biswas. Bayesian fault detection and diagnosis in dynamic systems. In *Proc. of the 7th Nat. Conf. on Artificial Intelligence*, pages 531–537. AAAI Press/The MIT Press, Cambridge, MA, 2000.

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

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

[18] S. Nouyan and M. Dorigo. Chain based path formation in swarms of robots. In *Ant Colony Opt. and Swarm Intelligence: 5th Int. Workshop, ANTS 2006*, volume 4150 of *LNCS*, pages 120–131. Springer Verlag, Berlin, Germany, 2006.

[19] S. Nouyan, R. Groß, M. Bonani, F. Mondada, and M. Dorigo. Group transport along a robot chain in a self-organised robot colony. In *Intelligent Autonomous Systems 9, IAS 9*, pages 433–442. IOS Press, Amsterdam, The Netherlands, 2006.

[20] R. O'Grady, R. Groß, F. Mondada, M. Bonani, and M. Dorigo. Self-assembly on demand in a group of physical autonomous mobile robots navigating rough terrain. In *Adv. in Artificial Life: 8th Euro. Conf., ECAL 2005*, pages 272–281. Springer Verlag, Berlin, Germany, 2005.

[21] R. Patton, F. Uppal, and C. Lopez-Toribio. Soft computing approaches to fault diagnosis for dynamic systems: A survey. In *Proc. of 4th IFAC Symp. on Fault Detect., Superv. and Safety for Tech. Processes*, pages 298–311. Elsevier, Oxford, UK, 2000.

[22] S. Roumeliotis, G. Sukhatme, and G. Bekey. Sensor fault detection and identification in a mobile robot. In *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 1383–1388. IEEE Press, Piscataway, NJ, 1998.

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

[24] V. Trianni and M. Dorigo. Self-organisation and communication in groups of simulated and physical robots. *Biological Cybernetics*, 95:213–231, 2006.

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

[26] V. Verma, G. Gordon, R. Simmons, and S. Thrun. Real-time fault diagnosis. *IEEE Robotics & Automation Mag.*, 11(2):56–66, 2004.

[27] V. Verma and R. Simmons. Scalable robot fault detection and identification. *Robotics & Autonomous Sys.*, 54(2):184–191, 2006.

[28] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. Lang. Phoneme recognition using time-delay neural networks. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 37:328–339, 1989.