Real-Time Diagnosis and Repair of Faults of Robot Control Software*

Gerald Steinbauer, Martin Mörth, and Franz Wotawa

Institute for Software Technology, Graz University of Technology
Inffeldgasse 16b/II, A-8010 Graz, Austria
{steinbauer, moerth, wotawa}@ist.tugraz.at

Abstract. Faults in hardware and software are not totally avoidable not even if the components are carefully designed, implemented and tested. In this paper we present a solution for detection, localization and repair of faults in the control software for autonomous mobile robots. The presented diagnosis system uses model-based diagnosis for fault detection and localization. Furthermore, we present a method which enables the robot control software to recover from located faults. The novelty of our approach is that fault localization and repair takes place at runtime. Moreover, we present experimental results of the proposed diagnosis system obtained in the RoboCup Middle-Size scenario.

1 Introduction

Even if the control software of a mobile robot is carefully designed, implemented and tested, there is always the possibility of faults in the system. Generally, faults are the deviation of the current behavior of a system from its desired behavior. For instance, we know very well situations in RoboCup Middle Size League (MSL) games where frequently robots had to be removed from and inserted into the field because some hardware or software components crashed or showed an undesired behavior. Carlson and Murphy [1] presented a quantitative evaluation of failures on mobile robots. The situation gets even worse if one thinks about autonomous robots, which operate for a long time without the possibility of human intervention, e.g. nuclear inspection robots, space probes or planetary rovers. Therefore, robustness and fault-tolerance are crucial for truly autonomous robots.

Because faults are not totally avoidable it is desirable that mobile robots are able to autonomously detect and repair such faults. If a permanent fault, e.g. broken hardware, is identified the robot should at least provide basic functionality or should be able to switch to a safe state. These requirements could be fulfilled if a dedicated diagnosis system is attached to the robot control software. Usually, a diagnosis system comprises three modules: (1) a monitoring module, (2) a fault detection and localization module and (3) a repair module. The first

* This research has been funded in part by the Austrian Science Fund (FWF) under grant P17963-N04 and Land Steiermark under grant 40Ro03-PE "RoboCup 2004".

A. Bredenfeld et al. (Eds.): RoboCup 2005, LNAI 4020, pp. 13–23, 2006.

module observes the actual behavior of the hardware and software of the robot system. The fault detection uses observations and a model of the system's desired behavior to detect deviations between them. A deviation is equivalent to a detected fault. However, in practice the detection of a fault is not enough. The module should also identify the hardware or software component which caused the fault. If a fault and its location is identified the repair module tries to resolve the fault. This could happen by a restart or reconfiguration of the affected components.

There are many proposed and implemented approaches for fault detection and repair in autonomous systems. The Livingstone architecture by Williams and colleagues [2] was used on the space probe *Deep Space One* to detect failures in the probe's hardware and to recover from them. The fault detection and recovery is based on model-based reasoning. Model-based reasoning uses a logic-based formulation of the system model and the observations. The advantage is that well understood reasoning algorithms can be used. Model-based diagnosis has also been successfully applied to fault detection in digital circuits, car electronics and software debugging of VHDL programs [3]. Verma and colleagues [4] used particle filter techniques to estimate the state of the robot and its environment. These estimations together with a dynamic model of the robot were used to detect faults. The advantage of this approach is that it accounts for uncertainties of the robot and its environment as it derives the most probable state. But so far it was only applied to fault detection in the robots hardware and no repair actions were derived. Rule-based approaches were proposed by Murphy and Hershberger [5] to detect failures in sensing and to recover from them. Additional sensor information was used to generate and test hypotheses to explain symptoms resulting from sensing failures. Roumeliotis et. al. [6] used a bank of Kalman filters to track specific failure models and the nominal model. The filter residuals were post-processed to produce a probabilistic interpretation of the system's operation. Such methods are popular for linear systems affected by Gaussian noise.

Up to now solutions for fault detection and repair of robot control software at runtime are rare. Most previous research has dealt either with hardware diagnosis or diagnosis of software as part of the software engineering cycle. In [7] Melchior and Smart summarized ideas and requirements for robots that are aware of failures at runtime. In this paper we present a solution for real-time fault detection and repair of control software of autonomous robots. The fault diagnosis follows the model-based diagnosis paradigm [8]. It is based on observations of the current behavior of the control system's components, a model of the desired behavior of the control system's components and the dependencies between them. A monitoring module constantly observes the behavior of the control software. If a deviation of the desired behavior is observed a diagnosis kernel derives a diagnosis, i.e., a set of malfunctioning software components explaining the deviation. Based on this diagnosis and a model of the software components and their connections a repair module executes an appropriate repair action to recover the system from the fault. The proposed diagnosis system has been

implemented and tested on our RoboCup MSL robots within the robotic soccer scenario. In the next section we describe the control software of our mobile robots. In section 3 we present the diagnosis system in more detail. Section 4 reports the results of experiments we conducted for the diagnosis system on our MSL robots. Finally, we draw some conclusions and give an outlook on future research.

2 Robot Control Software

The control software of our robots comprises different separated modules, called services. Each service runs as an independent process and implements a specific task, e.g., image processing, world modeling, planning. The control software is based on the MIRO framework [9]. MIRO is a CORBA-based software framework for robotics applications and provides a wide range of mechanisms for implementing services and for the communication between services. It is a very flexible framework which can be easily adapted to different robot platforms and applications [10].

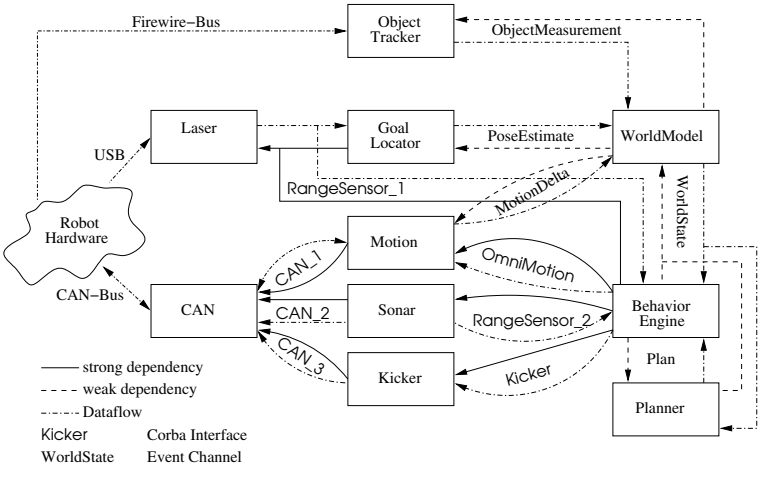

Fig. 1. Dependencies and data-flow within the robot control software

Figure 1 shows an overview of our robot control software. The software is organized in three levels with increasing abstraction. The Laser and CAN services provide low-level connection to the hardware of the robot. The connections are based on raw sensor data and low-level commands for actuators, e.g., laser scanner and the hardware modules on the CAN-Bus. The planner is located on top of the hierarchy and implements an abstract symbolic representation of the knowledge of the robot and its decision making process. The remaining services form the continuous level. Herein, all the computation of sensor inputs and the control of actuators on a continuous level are implemented, e.g., image processing, sensor fusion, execution of actions.

We use two different methods for the interactions and connections between the services: (1) remote CORBA method calls and (2) an event channel. Remote CORBA calls follow the client/server paradigm. One service, called the server, implements a specific function and exports a corresponding interface, e.g., the control interface for the omni-directional drive. The client, the service which uses the exported function, remotely invokes a method call on the server. The return value of the call may contain queried data. In Figure 1 remote CORBA calls are shown as solid lines directed to the server. The data-flow between server and client is shown as chain dotted lines. The Figure also shows the dependencies between services. Remote CORBA calls are called *strongly dependent* because a fault in the server directly affects the client.

The latter communication mechanism is the event channel. A server posts data via an event. All clients which are subscribed to this specific event are automatically informed if a new event is available. This connection type is shown as dashed lines and the data-flow direction is always directed from the server to the client. The event channel is called *weakly dependent*. The distinction between strong and weak dependencies is later important for the diagnosis and repair process.

The above described structure of the control software, the different types of connections and the dependencies between the services are used to build a model of the desired behavior of the control software. In the next section we describe how we use this model together with various observations of the behavior of services and connections to form a diagnosis system for the control software of our robots.

3 Diagnosis System

3.1 Monitoring

The task of the monitoring module of the diagnosis system is to observe the actual behavior of the control system. For this purpose we introduce the concept of observers. An observer monitors the behavior of a service or the communication between services. The observer determines a misbehavior of the control system if the observed behavior deviates from the specified behavior.

In the current implementation we use the following observers:

- *Periodic event production*: This observer checks whether a specific event e is at least produced every n milliseconds. An example for this observer is the event *MotionDelta* containing odometry data which is produced every 50 ms by the *Motion* service.
- *Conditional event production*: This observer checks whether an event e_1 is produced within n milliseconds after an event e_2 occurred. An example for this observer is the event *WorldState* which is produced by the *WorldModel* after an event *ObjectMeasurement* occurs.
- *Periodic method calls*: This observer checks whether a service calls a remote method m at least every n milliseconds. An example for this observer is the

RangeSensor interface of the *Sonar* service which is regularly called by the *BehaviorEngine*.
- *Spawn processes*: This observer checks whether a service spawns at least n threads. We know for instance if the *Motion* service works correctly it spawns six threads.

There are several requirements for the monitoring module. First, if observers are used there should be no or at least only a minimum necessity for changes in the control system. Furthermore, the monitoring component should not reduce the overall performance of the control system. Both requirements can be fulfilled easily by using mechanisms provided by CORBA [11] and the Linux OS. The first two observers are implemented using the CORBA event channel. The third observer is implemented using the CORBA portable interceptor pattern. The last observer is implemented using the information provided by the *proc* file-system of the Linux OS. For all these observers no changes are necessary in the control system. Furthermore, the computational power requirements for all the observers are negligible.

3.2 Diagnosis

A fault is detected if a observer belonging to the monitoring module recognizes a deviation between the actual and the specified behavior of the system. But so far we do not know which misbehaving service causes this fault. We use the model-based diagnosis (MDB) paradigm [8, 12] to locate the faulty service.

We will explain the principles of MDB with a simple example.

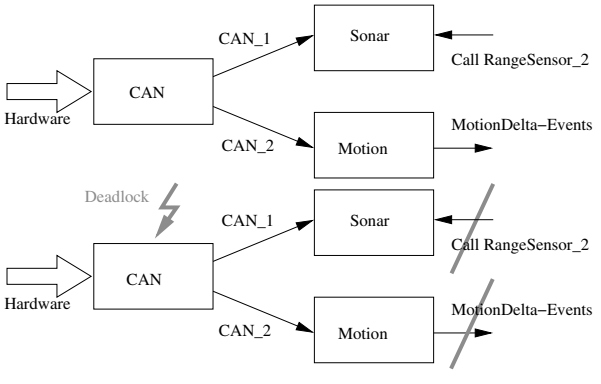

Fig. 2. Diagnosis of a fault in the Can-Service. The upper figure shows the desired behavior. The lower figure shows the behavior after a deadlock in the Can-Service.

Figure 2 shows an example for the diagnosis process in case of a malfunctioning CAN-Service. First, we build an abstract model of the correct behavior of the CAN-, Sonar and Motion Service. Therefore, we introduce two predicates: $AB(x)$ becomes true if a service x is abnormal, meaning x is malfunctioning.

$ok(y)$ becomes true if a connection y, either a remote call or an event, shows a correct behavior. The model for the correct behavior of the example could be described in the following clauses:

1. $\neg AB(CAN) \rightarrow ok(CAN\_1)$
2. $\neg AB(CAN) \rightarrow ok(CAN\_2)$
3. $\neg AB(Sonar) \wedge ok(CAN\_1) \rightarrow ok(RangeSensor\_2)$
4. $\neg AB(Motion) \wedge ok(CAN\_2) \rightarrow ok(MotionDelta)$

Lines 1 and 2 specify that if the CAN-Service works correctly also the connections *CAN_1* and *CAN_2* work correctly. Line 3 specifies that if the Sonar Service and its input connection *CAN_1* work correctly also the connection *RangeSensor_2* has to show a correct behavior. Line 4 specifies similar facts for the Motion Service.

If there is a deadlock in the CAN-Service, the Motion and Sonar services can not provide new events or calls as they get no more data from CAN. This fact is recognized by the corresponding observers and can be expressed by the clause: $\neg ok(RangeSensor\_2) \wedge \neg ok(MotionDelta)$. If we assume a correct behavior of the system expressed by the clause $\neg AB(CAN) \wedge \neg AB(Sonar) \wedge \neg AB(Motion)$ we get a contradiction. This means we have detected a fault.

Finding the service which caused the fault is equivalent to finding the set of predicates $AB(x)$ with $x \in \{CAN, Motion, Sonar\}$ that resolves the contradiction. These sets are called *diagnoses*, Δ. We are interested in finding a set with minimal cardinality, e.g., a single faulty service. These diagnoses are in general sufficient as multiple faults are unlikely. In this example the set $\{AB(CAN)\}$ with only one element is able to resolve the contradiction. Therefore, the faulty CAN-Service is located.

3.3 Repair

Once the diagnosis system has found one or more malfunctioning services responsible for a detected fault it should be able to recover the control system from this fault. Therefore, the repair module determines an appropriate repair action based on the described diagnosis and the dependencies between the services.

The repair action comprises a stop and a restart of the malfunctioning services. But we have to be careful, because restarting a specific service may also cause a restart of other services depending the restarted service. Therefore, the repair module takes the *strong dependencies* between services into account.

The appropriate repair action is derived in the following way: Put all members of the diagnosis Δ in a set R. The members of this set R are scheduled for restart. In the next step insert all services into R, which strongly depend on a member of R. Repeat this step until no more services are added to R. R now contains all services which have to be restarted. But first of all the scheduled services have to be stopped in an ordered way. This means to first stop all services which no other service strongly depend on. Afterwards, stop all services for which no more services are running which depend on them. This process is necessary to avoid additional crashes of services caused by a sudden stop of a service another service

depends on. Hereafter, start all affected services in the reverse order. Services which were restarted because of a strong dependency on the malfunctioning services should be able to retain data which it had gained so far or they should be able to recover such data after a restart. Otherwise the control system may become inconsistent.

After this repair action took place, the robots control system is again in the desired state.

4 Experiments

The proposed diagnosis system has been implemented and tested on the robots of our RoboCup MSL team. The robot control system runs on an embedded Pentium III PC with 850 MHz clock rate and 256 MB of RAM. The operating system on the PC is an ordinary Linux system.

The diagnosis system itself is implemented as a separate process to minimize the interference with the existing control system. The diagnosis system implements the four types of observers described in Section 3. The use of Corba and OS services allows monitoring of the robot control system without any impact to it.

The model of the robot control system (software components, dependencies, observers) is specified in a XML file. Therefore, changes in the model or adaptation to other software systems are simple and straight forward. Table 1 shows a section of the specification of the used model describing the behavior of the Motion service.

Table 1. Model description for the motion service

```
<component id="MotionService">
  <rule class="SPAWNS-PROCESS-RULE">
    <property name="min-process-count">6</property>
  </rule>
  <rule class="DIRECT-DEPENDENCY-RULE">
    <property name="component-id">CanService</property>
  </rule>
  <rule class="PERIODIC-EVENT-PRODUCTION-RULE">
    <property name="max-sleep-time">150</property>
    <property name="event-name">MotionDelta</property>
  </rule>
</component>
```

The diagnosis system is divided into three modules: (1) a monitoring module, (2) a diagnosis kernel and (3) a repair module. The monitoring module starts all necessary observers according to the model description and regularly checks for violations of the observers. If such a violation is detected the diagnosis kernel is informed. The diagnosis kernel derives a diagnosis based on the model of the control system and the violated observations. The derivation of a diagnosis

is started after a certain amount of time, i.e. 5 s, within no more changes in the states of the observers are detected. This is done for stability reasons as it takes a certain amount of time for all observers to recognize an improper behavior. The diagnosis will be communicated to the repair module. It executes the appropriate repair action to recover the control system. During the repair action no new diagnoses are derived. We do this for stability reasons as the repair action temporally may violate additional observers. After the repair action is completed the observers and the diagnosis kernel are started again.

For the evaluation of the proposed diagnosis system and its implementation we did several experiments on our mobile robots. We introduced artificial faults into the robot control system and analyzed if the diagnosis system detected and located the fault and recovered the control system. We used two different fault scenarios:

- *Killing a Service*: A certain software service is explicitly killed. This is equivalent to a crash of a certain service.
- *Deadlock a Service*: A deadlock is introduced to a certain software service. This is equivalent to a malfunctioning software service.

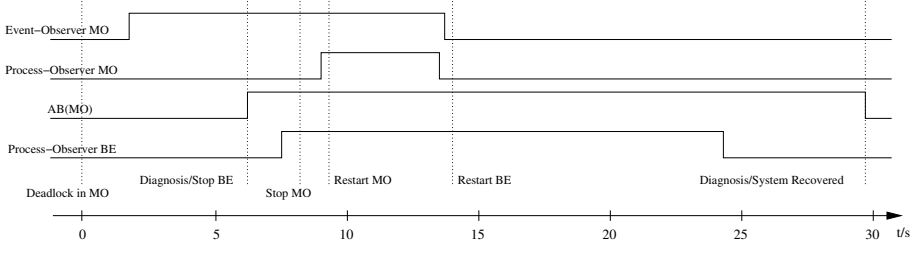

Fig. 3. Timing diagram for diagnosis and repair of a deadlock in the motion service

Figure 3 shows the timing diagram for the diagnosis and repair of an introduced deadlock in the motion service (MO). After introducing the deadlock in MO the Periodic Event Observer for the event *MotionDelta* detects that no more events are produced. After the waiting time the diagnosis kernel derived that MO is malfunctioning, *AB(MO)*. Instantly the repair process starts. The repair action comprises a stop of the Behavior Engine (BE), a stop of MO, and a restart of MO and BE. The restart of BE is necessary because BE strongly depends on MO. Again after the waiting time the diagnosis kernel derives the diagnosis that all services work properly now. Please note that no other services were affected by the repair process. The Figure also shows the fact that suspending the diagnosis kernel during the repair is necessary as observers report additional improper observations, e.g. Process Observer. The relatively long time for the recovery can be explained by the fact that stopping and starting of services can take a while because of the required starting, stopping and re-configuration of hardware components. Furthermore, stopping a service may consists of various

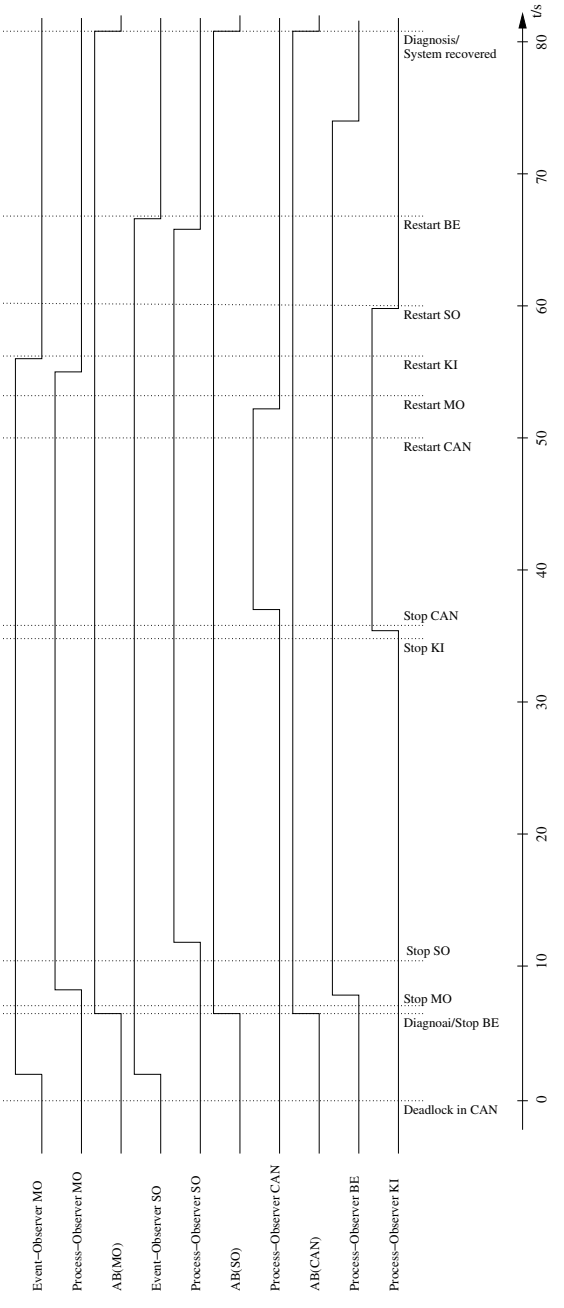

Fig. 4. Timing diagram for diagnosis and repair of a deadlock in the CAN service

steps. First, we send the service a interrupt signal (SIGINT). This allows a service to shutdown properly. If this does not stop the service within 5 seconds we send the service a termination signal (SIGTERM). Finally, we send a killing signal (SIGKILL) if after another 5 seconds still processes of the services run. The time for computing the diagnosis is negligible because it is less than 10 ms.

Figure 4 shows a more complex scenario. Here we introduce a deadlock in the CAN-Service. After introducing the deadlock, MO and the Sonar Service SO produce no more data because they get no more data from CAN. This fact is detected by the appropriate observers. Using the model of the observations, the components and its connections the diagnosis kernel recognizes the malfunctioning CAN. The repair action is similar to the example above except that more services are involved. After repair, the control system is again in the desired state.

We conducted also two experiments in which we killed a service. In the first experiment we killed the Laser Service. The diagnosis system successfully detected the fault and recovered the control system by restarting BE, Goal Locator and Laser. The recovery took 68 s. In a second experiment we killed the World Model. The diagnosis system successfully detected and repaired the fault. During this experiment it was important that the whole process took only 20 s because the system located the fault in the WM and no other service was affected.

We also tested the diagnosis system during games of a RoboCup MSL exhibition at our university. During the games the diagnosis system performed well. The image processing of one of our robots crashed twice during the games because of problems with a new camera. But the diagnosis system successfully detected, localized and repaired the fault at runtime.

The affect of the diagnosis system on the runtime performance of the robot control system is negligible. The diagnosis system uses less than 1 % of the CPU time and less than 5 % of the memory.

5 Conclusion and Future Work

In this paper we presented a diagnosis system capable of real-time fault detection, localization and repair for the control software of autonomous mobile robots. The proposed system follows the model-based diagnosis paradigm. It uses a general abstract model of the correct behavior of the control system together with observations of the actual behavior of the system to detect and localize faults in the software. Furthermore, we presented a repair method which is able to recover the software from a fault. Because of its general methods the proposed system is also applicable to other software than robot control software.

The proposed diagnosis system has been successfully implemented and tested on our RoboCup MSL robots. Experiments show that the system is able to detect and localize faults like crashed or deadlocked services at runtime. Furthermore, the system is able to recover the control software from such faults on the fly. Moreover, the proposed diagnosis system can be deployed on a robot platform with no changes in the existing control software and with nearly no effect on the runtime performance of the control system.

For future research it would be interesting to improve the used models by using more knowledge about the robot and its environment. Therefore, diagnosis of more complex systems and also the robots hardware would be possible. Another interesting issue would be the automatic reconfiguration of the robots hardware and software to recover from permanent faults, like broken hardware.

References

1. Jennifer Carlson and Robin R. Murphy. Reliability Analysis of Mobile Robot. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation, ICRA 2003, September 14-19, 2003, Taipei, Taiwan*. IEEE, 2003.
2. B. C. Williams, P. Nayak, and N. Muscettola. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.
3. Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, 1999.
4. V. Verma, G. Gordon, R. Simmons, and S. Thrun. Real-time fault diagnosis. *IEEE Robotics & Automation Magazine*, 11(2):56 – 66, June 2004.
5. Robin R. Murphy and David Hershberger. Classifying and recovering from sensing failures in autonomous mobile robots. In *AAAI/IAAI, Vol. 2*, pages 922–929, 1996.
6. S.I. Roumeliotis, G.S. Sukhatme, and G.A. Bekey. Sensor fault detection and identification in a mobile robot. In *IEEE Conf on Intelligent Robots and Systems*, pages 1383 – 1388, Victoria, Canada, 1998.
7. Nik A. Melchior and William D. Smart. Autonomic systems for mobile robots. In *Proceedings of the International Conference on Autonomic Computing (ICAC 2004)*, pages 280–281, 2004.
8. Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
9. Hans Utz, Stefan Sablatng, Stefan Enderle, and Gerhard K. Kraetzschmar. Miro – middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, 18(4):493–497, August 2002.
10. Gordon Fraser, Gerald Steinbauer, Arndt Mühlenfeld, and Franz Wotawa. A modular architecture for a multi-purpose mobile robot. In *Proc. of the 17th Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, volume 3029 of *Lecture Notes in Artificial Intelligence*. Springer, 2004.
11. Michi Henning and Steve Vinoski. *Advanced CORBA©Programming with C++*. Addison Wesley Professional, 1st edition, 1999.
12. Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.