Multi-agent-based Diagnostics of Automotive Electronic Systems

Dušan Pavlíček, Michal Pěchouček, Vladimír Mařík, and Ondřej Flek

Gerstner Laboratory, Czech Technical University in Prague,
Technická 2, 166 27, Prague 6, Czech Republic
pechouc@labe.felk.cvut.cz

Abstract. The diagnostic system discussed in this paper provides an agent-based approach to advanced on-board diagnostics of operation and communication failures occurring in an electronic automotive system. To facilitate development and testing of agent-based automotive diagnostics, a software simulator of automotive electronics systems, also based on multi-agent technology, has been implemented, allowing the developers to simulate various automotive system failures. The paper presents both the proof-of-concept on-board diagnostics system and the simulation layer used for its development and testing. The following topics are covered: simulation of the hardware units, simulation of user-invoked hardware failures, distributed detection and diagnostics of the failures, and visualization. The proposed automotive diagnostics approach can be deployed on real automotive electronics hardware.

1 Introduction

Graceful degradation or *fault tolerance* in terms of automotive electronic systems is the ability of the system to classify all detected failures occurring in the system according to their seriousness and potential impact and selectively disable only the affected parts of the system while the rest of the system can remain in operation. Prediction of future propagation of the detected failure across the electronic system may also be involved in this process. Such behavior allows far more precise and adequate reaction to failures than just one-shot disabling of the entire system.

Effective estimation of the severity of system failure and of the possible impact and future development of the problem also allows the diagnostic system to present the vehicle driver with relevant information at the right moment, ensuring safe operation of the vehicle on one hand, and avoiding unnecessary distraction and/or overburdening of the driver with excessive information on the other.

The approach discussed in this paper uses multi-agent technology to address this issue. An on-board automotive diagnostics system based on such approach is presented. Since developing such system directly on real-world automotive electronics hardware would not be economically acceptable, a test-bed capable of simulating the functionality of automotive systems and modeling of their various

failures had to be implemented. Multi-agent technology proved very efficient in this task, too. Thus, the proof-of-concept on-board automotive diagnostics system and the simulation test-bed are each implemented by a separate layer of agents. The two layers are very loosely coupled. It is assumed that if the agent-based approach to automotive diagnostics proves viable, the simulation layer could be replaced by real-world automotive electronics and the diagnostics layer deployed to such hardware without significant modifications.

The key strengths of multi-agent systems are their distributed nature and parallelism. This makes them particularly suitable for application in the field of simulation of complex electronic systems, as well as their diagnostics. Another advantage of multi-agent systems in general is their modularity and natural support for reconfiguration. This provides the diagnostic layer with the ability to reconfigure itself dynamically to reflect a changed configuration of the diagnosed system. In the simulation layer, dynamic loading and unloading of selected hardware devices at run-time upon user's request is possible, facilitating simulation of various hardware configurations without the need to restart the system.

The fact that the actual hardware layer is simulated by software agents is highly beneficial for many reasons. The user has full control over simulation of failures occurring in the system and thus can simulate various failure scenarios. Thanks to the agent introspection and easy ad-hoc modification available to the user, the simulation layer can serve not only as a powerful platform for development and testing of automotive diagnostics systems but, as a by-product, can also be used as an advanced test-bed for a number of other tasks, including hardware performance and stability testing, behavior analysis, internal hardware algorithm design and optimization etc.

The discussed proof-of-concept diagnostic system involves four types of failure diagnostics, including prediction of both direct and indirect immediate influence of the detected failures on individual hardware units in the system. Two major types of failures can be detected: failures affecting output signals of hardware units (i.e. their internal failures) and failures of communication links between the units.

2 System Overview

The presented software system consists of four major parts:

- entity execution manager (see section 3)
- simulated hardware layer (see section 4)
- diagnostic layer (see section 6)
- visualization module (see section 7)

The system is built on top of the A-globe [1], [4] multi-agent platform which provides infrastructure for basic agent functionality (environment simulation, yellow pages services, message transport etc.), and A-globeX Simulation layer responsible for executing predefined scenarios, managing central virtual simulation clock updates and so forth. For portability reasons as well as for fast

development and easy maintenance purposes, the entire system is written in Java.

All entities in the system (simulated hardware units, diagnostic agents, visualization module, internal system modules etc.) have the form of autonomous agents. There are currently approx. 180 agents running in the implemented scenario (a door-locking electronic subsystem). The entire multi-agent system is fully distributed and asynchronous by its nature. Agents in A-globe "live" on containers, i.e. software platforms that provide their "inhabitants" with basic agent-oriented infrastructure.
There are two types of containers:

– a single central **master container**, populated by system agents including the Visio Agent as well as agents responsible for simulation of the scenario environment and flow of time, particularly the Car Agent.
– multiple **client containers**, each of them hosting a single scenario agent, either a simulated hardware unit (electronic control unit – ECU, sensor, switch, BUS) or a diagnostic agent (monitoring, data flow, data consistency, prediction).

Each container can run on a separate computer and therefore all agents communicate strictly by using messages sent through the network. The message transport layer is provided to the agents by the individual A-globe containers. In order to find each other in the network, agents use directory services ("yellow pages") provided also by the containers.

Communication with the server container (environment simulation and visio) is performed by a special type of system messages called *topics*. Unlike standard messages, topics can be sent only from the server to client containers or vice versa, but never between client containers only. Topics can also be broadcast, e.g. when the server container sends system time updates to client containers.

3 Dynamic Scenarios

A particular simulation scenario to be executed (i.e. startup configuration of simulated hardware units and diagnostic agents) is created dynamically. Static XML files hold only information about all usable agent types, including their global parameters. All the rest of the process of scenario loading (and possible subsequent modification) is controlled by the user at run-time. The Entity Execution Manager allows the user to choose interactively which simulated hardware units will be loaded when the given scenario is executed (by pressing the Start Scenario button in the Entity Manager).

Moreover, upon user's request selected simulated hardware units can also be loaded or unloaded dynamically during run-time. The Entity Execution Manager handles loading and unloading of all corresponding diagnostic agents automatically.

This feature can be used e.g. for dynamic reconfiguration of the simulated system: adding new hardware units or introducing alternative versions of particular hardware units during run-time.

4 Simulated Hardware Layer

All hardware units (electronic control units – ECUs, sensors, switches and BUSes) are simulated by dedicated agents. One type of agent is used for simulation of BUS units, while another type is used for simulation of ECUs, switches and sensors.

Note that if the diagnostic system is deployed on real hardware, the entire simulated hardware layer would be replaced by actual hardware devices, similarly as in [2], [3].

Communication between hardware units is simulated by sending messages among corresponding agents. Each transmitted signal is physically represented by a message sent between the corresponding agents.

While the common functionality of all agents simulating hardware devices is implemented by the same code, internal behavior of each ECU, switch or sensor is defined by a reference to algorithm specific for each hardware unit. Such algorithm defines responses to particular incoming signals depending on the current internal state of the given hardware unit. Internal state of a simulated hardware unit is stored in a set of internal variables, e.g. `VarBool`, `VarInt`, `VarCounter` etc. that are involved in the internal algorithms (i.e. body) of the simulated hardware units. The advantage of these variables over standard built-in Java variables (`boolean`, `int` etc.) is that any change of their value can trigger an event and thus they can be monitored easily. Also, their value can be imported and exported which makes it possible to import and export the current state of the entire simulated hardware unit. Time dependencies within the internal algorithms are handled by so called timers (`VarTimer`), internal variables that expire after a predefined period of time. When timer expiration occurs, the internal algorithm is notified about it and it can take an appropriate action.

Simulated hardware units not only communicate with each other via signals but some of them must also react to changes of their environment. For example, sensors repetitively read values of physical phenomena they are dedicated to, and almost all simulated hardware units need to be aware of flow of simulated time. The real-world environment is simulated by a central Car Agent. This agent not only handles the flow of central simulation time but also keeps information of the current physical status of all car parts related to the scenario.

Interaction of hardware units with the environment is implemented by sending a special type of system messages from the central Car Agent to the hardware agents and vice versa if needed (e.g. Door Control Units physically moving door lock motors).

The topology of the network of hardware devices is defined statically. For each signal, one or more sender-receiver pairs of units are defined. This way, the entire system topology is fully described. As a by-product, a number of other types of information can be derived from this elementary description, such as all input and output signals for a particular hardware unit. These automatically derived types of data are subsequently used for diagnostic and visualization purposes.

5 Failure Simulation

In order to provide the various diagnostic agents with relevant data to operate on, it was necessary to allow the user to introduce failures into the simulated hardware system.

All failure types are simulated within the simulated hardware layer, more specifically inside hardware unit agents. Therefore diagnostic agents are not directly aware of any failures introduced by the user and they have to detect them by actual observation of behavior of the hardware system they are deployed on.
The user can introduce two main types of failures:

- failures of **hardware units** (signal drop-out, invalidation, alteration)
- failures of **communication links** between devices (signal drop-out)

Failures of a hardware unit are implemented in the output module of the agent. After the internal logic module of the agent generates an outgoing signal, this signal can be either dropped, invalidated (wrong data format) or altered (modified data value) by the output module before it is actually sent through the network, depending on the user's input. Notice that the internal logic module is unable to influence this process.

Failures of communication links are implemented in the input module of the agent. Before an incoming signal is processed by the internal logic module of the agent, it can be dropped by the input module, depending on the user's input. If a signal is dropped, it never reaches the internal logic module and therefore it is never actually processed. Again, notice that the internal logic module is unable to influence this process.

6 Diagnostic Layer

There are currently four types of agent-based diagnostics implemented:

- Monitoring Layer (see section 6.1)
- Data Flow Layer (see section 6.2)
- Data Consistency Layer (see section 6.3)
- Prediction Layer (see section 6.4)

They all observe the same underlying simulated hardware system from different perspectives and provide the user with a wide range of diagnostic results.

6.1 Monitoring Layer

Each hardware unit (simulated by an agent on the hardware layer) is accompanied by one dedicated Monitoring Agent.

Whenever the Entity Execution Manager loads a new simulated hardware unit to the system, it also loads a corresponding Monitoring Agent.

The Monitoring Agent possesses a reference model of behavior of the hardware unit it monitors. For easy code management and development efficiency reasons, the current implementation of the reference model uses the same algorithms (and even the same source code classes) as the actual simulated hardware units.

If the diagnostic system was deployed on real hardware and the agents on the simulated hardware layer were replaced by actual hardware devices, these reference behavior algorithms would have to be updated in order to match the behavior of the real hardware as closely as possible.

The Monitoring Agent "listens" to all input and output signals of the hardware unit it monitors, it processes all the input signals with the reference algorithm and it compares its results with the actual output signals that the monitored hardware unit produced in response to the given input signals. If any differences between the real hardware and the reference model are detected, they are classified as follows:

- signal **drop-out** – the reference algorithm generated an output signal but the hardware unit did not
- signal **invalidation** – an output signal was generated by the hardware unit but the signal data is in wrong (illegible) format
- signal **alteration** – both the reference algorithm and the hardware unit produced an output signal but the signal data values differ

It is obvious that the Monitoring Agent and the monitored hardware unit must be well synchronized for this diagnostic method to provide reliable results.

Apart from the forementioned functionality, the Monitoring Agent also detects whether the monitored hardware unit receives and sends all signal types regularly as expected. If a particular signal type is not received or sent by the monitored hardware unit within a predefined period of time, a signal timeout is detected and classified as either input or output signal drop-out.

6.2 Data Flow Layer

While the Monitoring Agents are responsible for observing the operation of hardware units, the Data Flow Agents focus on reliability of communication links between hardware units.

Each Data Flow Agent is dedicated to a particular signal type.

Whenever the Entity Execution Manager loads a new simulated hardware unit into the system, it also loads all Data Flow Agents responsible for monitoring all input and output signal types of that hardware unit, unless the particular Data Flow Agents have been loaded previously together with other hardware units.

A Data Flow Agent assigned to monitor the flow of a particular signal type "listens" to all signals of that type sent through the hardware network. The Data Flow Agent checks whether all sent signals (of the particular type) reach

their destination successfully. If a sent signal is not received within a predefined time period, a signal timeout is detected and reported as a failure of the communication link between the sender and the receiver.

Note that if the sender stops generating the output signal as a result of internal failure, the Data Flow Agent will not pay attention to it. This is the responsibility of the corresponding Monitoring Agent.

6.3 Data Consistency Layer

Consistency Agents check whether a certain type of information, or *property* (carried by a certain type of signal), that is shared among several hardware units, is consistent across all of these units.

Each Consistency Agent is dedicated to monitoring consistency of a certain property, i.e. certain type of signal across the entire system.

Whenever the Entity Execution Manager loads a new simulated hardware unit to the system, it also loads all Consistency Agents responsible for monitoring all properties (represented by input and/or output signals) of that hardware unit, unless the particular Consistency Agents have been loaded previously together with other hardware units.

If any inconsistency of a particular property is detected, the Consistency Agent uses the static topology definition of the hardware system to generate description of the hardware unit sequence through which the property is propagated and identifies all hardware units whose property value differs from the value of the first unit in the sequence (the originator), e.g. the sensor. The Consistency Agent also compares property values of all pairs of adjacent hardware units to identify the exact locations where the change of the propagated property value – i.e. failure – occurred. This information can also be used to classify the nature of the failure: if the value change occurs between the input and output of the same hardware unit, it is clear that we are dealing with a failure of the actual hardware unit; if the value change occurs between the output and input of two different hardware units, it indicates that the failure is located on the way between these units, most likely within their communication link.

The advantage of this diagnostic method is that Consistency Agents don't need any knowledge of internal behavior of the observed hardware units and they regard them as black boxes.

6.4 Prediction Layer

The prediction diagnostic layer continually receives reports of failures detected by other diagnostic layers and generates prediction on possible direct or indirect influence of the detected failures on other parts of the entire hardware system.

This type of diagnostics can be called *forward diagnostics* as it simulates spreading of failures in a forward direction through the system.
This diagnostic layer consists of two major parts:

- a single Prediction Coordinator

– multiple Prediction Units – one agent for each ECU

When a Monitoring Agent (monitoring diagnostic layer) or a Data Flow Agent (data flow diagnostic layer) detects a failure in the hardware system, it notifies the Prediction Coordinator. The failure report contains the following items: signal name, signal sender and receiver, an *isActive* flag and a *failureID* (used internally by the prediction diagnostic layer).

If the reported failure is active and is new for the Coordinator (i.e. it is not being processed already), the Coordinator assigns a unique ID to the failure and activates the Prediction Unit that corresponds to the receiver of the faulty signal.

Each Prediction Unit contains the same model of the ECU's internal behavior as the corresponding Monitoring Agent. When the Prediction Unit is activated, it starts to receive regular internal state updates from the respective Monitoring Agent and it applies them to its own copy of the ECU's internal logic model, thus keeping it up-to-date.

Apart from activating the Prediction Unit of the receiver ECU, the Coordinator also sends this agent a copy of the failure report for processing.

Upon receiving a failure report, the Prediction Unit runs various instances of the faulty signal through the ECU's internal behavior model. The signal instances contain different data values, depending on the data type of the signal:

– boolean – `true`, `false`
– three-state boolean – `true`, `false`, `undefined`
– integer – a few selected numeric values representatively covering the entire range of data values of the particular signal, e.g. minimal and maximal value etc.

It is crucial that before each input signal is processed, the internal behavior model is reset to a state corresponding to the latest state update received from the respective Monitoring Agent.

The Prediction Unit agent checks whether any of the processed input signal instances results in generation of any output signal. Depending on the results of this procedure, the Prediction Unit agent sends the Coordinator all output signals of the ECU's internal behavior model that are marked either as active or inactive, depending on the fact whether they were or were not generated in response to the particular failure (identified by a unique ID).

If an output signal is marked as active in this step, it means it may be influenced by the particular faulty input signal and thus may also become faulty.

Note that all currently active failures (i.e. faulty input signals) are also processed whenever the Prediction Unit receives a new state update from the Monitoring Agent.

When the Prediction Coordinator receives results from any of the Prediction Unit agents, it handles them in a very similar way to how it handles failure reports received from any diagnostic agent. It sends them to the corresponding Prediction Units (signal receivers) as a failure report. This way the potential failure iteratively spreads through the system.

When a particular failure is deactivated, i.e. the Prediction Coordinator receives a failure report from a diagnostic agent having the `isActive` field set to `false`, the Coordinator tries to find the received failure report in its list of currently active failures and thus identify the (previously assigned) ID of the failure. When the failure ID is found, a report on this failure's deactivation is propagated through all involved Prediction Units.

7 Visualization Module

The visualization module is implemented as an agent located on the server container. It provides the following modules and functionality:

- interactive GUIs allowing the user to interact with the hardware system
- hardware system topology viewer
- signal sniffer (filterable via the filter tree)
- tools for failure control
- visualization of diagnostic results

8 Use Case

This section provides a brief demonstration of the diagnostic functionality. Introduction of a simple failure to the simulated automotive system and the response of the diagnostic layer is shown.

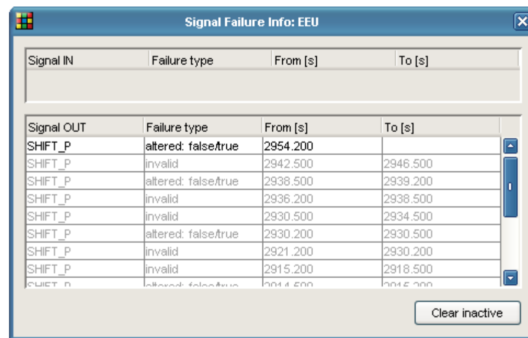
Let us choose the `SHIFT_P` signal for now. This signal is generated by the *Shift Switch* unit, then it is sent to the *Engine Electronics Unit* (*EEU*) which passes it on to the *Electronic Ignition Lock* unit (*EIL*).

We set both the invalidation and alteration probability rate of the `SHIFT_P` output signal of the *EEU* to 50 percent.

Once the failure parameters are applied, we can use the signal sniffer window to observe the randomly invalidated or altered `SHIFT_P` signals generated by the *EEU*.

Monitoring agents dedicated to observation of the *EEU* and *EIL* units start to detect the randomly occurring signal failures at the output (*EEU*) or input (*EIL*) of the respective unit. This is indicated to the user by lighting up the corresponding red diagnostic "diodes" in the visualization of the two hardware units.

The user can also inspect detailed descriptions of the detected failures for each unit. Failures that are no longer active are displayed in gray.



All `SHIFT_P` signals transmitted within the system are also processed by the data consistency agent responsible for monitoring the `SHIFT_P` signals. Data value of each instance of the `SHIFT_P` signal is compared with the initial data value that was generated by the *Shift Switch* unit (set to `SHIFT_P = true`) before reaching the *EEU* and *EIL* units. Since the *EEU* sets the invalidated `SHIFT_P` output signals to `SHIFT_P = ???` and the altered `SHIFT_P` output signals to `SHIFT_P = false`, these invalidated or altered instances of the `SHIFT_P` signal are regarded as data inconsistencies when compared to the original `SHIFT_P` signals generated by the *Shift Switch* unit.

Detailed information about the detected data inconsistencies can be inspected in the *Data inconsistencies* window.

Data inconsistencies

Property	Sequence	Data	Inc. Entities	Inc. Pairs	From [s]	To [s]
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > ??? > ???	EEU(out), EIL(in)	EEU(in) > EEU(out)	2986.600	
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > ??? > ???	EEU(out), EIL(in)	EEU(in) > EEU(out)	2981.300	2984.300
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > true > ???	EIL(in)	EEU(out) > EIL(in)	2975.200	2975.300
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > true > false	EIL(in)	EEU(out) > EIL(in)	2972.200	2972.300
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > true > false	EIL(in)	EEU(out) > EIL(in)	2963.200	2963.300
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > true > ???	EIL(in)	EEU(out) > EIL(in)	2957.200	2957.300
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > true > ???	EIL(in)	EEU(out) > EIL(in)	2946.500	2946.600
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > true > false	EIL(in)	EEU(out) > EIL(in)	2939.200	2939.300
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > true > false	EIL(in)	EEU(out) > EIL(in)	2934.500	2934.600
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > true > ???	EIL(in)	EEU(out) > EIL(in)	2918.500	2918.600
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > true > false	EIL(in)	EEU(out) > EIL(in)	2909.200	2909.300
ShiftP	ShiftSwitch(out) > EEU(in) > EEU(out) > EIL(in)	true > true > true > ???	EIL(in)	EEU(out) > EIL(in)	2903.200	2903.300

Clear inactive

For the next part of the demonstration we will choose the `WHEEL_SPEED` signal. This signal is generated by four independent *Wheel Speed Sensors*. The four independent signals are all received by the *Active Body Control* unit (*ABC*) which uses them for calculation of the speed of the car. The resulting car speed is them sent to the *EIL* in the form of the `CAR_SPEED` signal.

We set the drop-out probability rate of the `WHEEL_SPEED1` input signal of the *ABC* to 100 percent. This setup can be regarded as a simulation of a broken communication link between the *Wheel Speed Sensor 1* and the *ABC*: `WHEEL_SPEED1` signals are sent from the *Wheel Speed Sensor 1* successfully but they are not received by the *ABC*.

The `WHEEL_SPEED1` signal lost on its way is detected by the data flow agent responsible for monitoring all `WHEEL_SPEED1` signals.

Also, the monitoring agent dedicated to the *ABC* detects that no `WHEEL_SPEED1` signals are coming from the *Wheel Speed Sensor 1* which is indicated to the user by lighting up the corresponding red diagnostic diode.

Let us now focus on the prediction diagnostic layer. Hardware units that are identified by this layer as potentially affected by the detected failure are marked by a lit-up orange diode. The prediction is updated with each change of state of the system.

Let us assume that the car is currently parked and the ignition is turned off. In such situation, the units identified as potentially affected are the *ABC* and *EIL*. The *ABC* does not currently receive any updates of the `WHEEL_SPEED1` signal, therefore it is directly affected by the failure we introduced earlier. Because of that, the speed of the car that the `ABC` calculates and sends to the *EIL* may also be incorrect. Therefore the *EIL* is indirectly influenced by the given failure.

If we now turn the ignition on, the state of the system changes, the prediction is updated and all *Door Control Units* (*DCUs*) are now also marked as potentially affected by the failure. The reason why this happens is that the *EIL* sends a request to all the *DCUs* to lock all the doors (signal `DOOR_DEMAND_LOCK`) when the ignition is on and the speed of the car exceeds 20 km/h. The ignition is on now but since the *EIL* may receive an incorrectly calculated car speed from the *ABC*, proper generation of the `DOOR_DEMAND_LOCK` signal may be influenced by this failure. Therefore all the *DCUs* are indirectly influenced now as well because they may not receive the `DOOR_DEMAND_LOCK` signal correctly.

9 Conclusion

Agent technology proved not only very promising in the area of diagnostics of automotive electronic systems, but also very efficient in simulation of such systems for the purpose of development and testing. The distributed multi-agent system is particularly useful for modeling various scenarios both in terms of online hardware reconfiguration and failure simulation. The natural modularity of the agent-based simulation and diagnostics is a key feature and benefit of the proposed solution. Also, the distributed and parallel approach to failure detection and diagnostics increases its robustness and enables easy and effective deployment of the described technology even in large-scale electronic systems.

The implemented failure prediction considers only the current state of the underlying hardware system. One of our future goals is to extend the implementation so that the prediction would also consider selected *future* states of the system and based on the results of the individual prediction passes. The system would then automatically suggest to the user what action to take in order to minimize the number of hardware devices influenced by the detected failure.

References

1. A-globe. A-globe Agent Platform. `http://agents.felk.cvut.cz/aglobe`, 2006.
2. F. P. Maturana, R. J. Staron, and K. H. Hall. Methodologies and tools for intelligent agents in distributed control. *IEEE Intelligent Systems*, 20(1):42–49, 2005.
3. G. M. Provan and Yi-Liang Chen. Agent-based, distributed diagnosis for shipboard systems. In *BASYS '02: Proceedings of the IFIP TC5/WG5.3 Fifth IFIP/IEEE International Conference on Information Technology for Balanced Automation Systems in Manufacturing and Services*, pages 281–288, Deventer, The Netherlands, The Netherlands, 2002. B. V. Kluwer.
4. D. Šišlák, M. Rehák, M. Pěchouček, M. Rollo, and D. Pavlíček. A-globe: Agent development platform with inaccessibility and mobility support. In R. Unland, M. Klusch, and M. Calisti, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 21–46, Berlin, 2005. Birkhauser Verlag.