Runtime Reflection: Dynamic model-based analysis of component-based distributed embedded systems

Andreas Bauer Martin Leucker Christian Schallhart

Institut für Informatik, Technische Universität München
{baueran, leucker, schallha}@informatik.tu-muenchen.de

Abstract

Distributed embedded systems have pervaded the automotive domain, but often still lack measures to ensure adequate behaviour in the presence of unforeseen events, or even errors at runtime. As interactions and dependencies within distributed automotive systems increase, the problem of detecting failures which depend on the exact situation and environment conditions they occur in grows. As a result, not only the detection of failures is increasingly difficult, but also the differentiation between the symptoms of a fault, and the actual fault itself, i. e., the cause of a problem.

In this paper, we present a novel and efficient approach built around the notion of a software component similar to AUTOSAR, for dynamically analysing distributed embedded systems in the testing phase or even in standard operation, in that we provide a framework for detecting failures as well as identifying their causes. Our approach is based upon monitoring safety properties, specified in a language that allows to express dynamic system properties. For such specifications so-called monitor components are generated automatically to detect violations of software components. Based on the results of the monitors, a dedicated diagnosis is then performed in order to identify explanations for the misbehaviour of a system. These may be used to store detailed error logs, or to trigger recovery measures.

1. Introduction

Handling the ever growing share of software in present-day cars poses an ongoing challenge to software engineering research. As a matter of fact, it is estimated that 30% of the value added in automotive is up to electronics comprising embedded system functions [16]. The networking of formerly autonomous functions causes difficult integration issues, and results in potentially error-prone development and deployment processes. The model-based design and development of embedded automotive systems, especially in safety-critical settings can be accompanied by the use of formal or semi-formal methods, such as simulation or model-based testing [5, 7, 4], in order to increase our confidence in the correctness of the system. However, such methods if employed in the design and development process alone cannot guarantee that systems are sufficiently prepared for dealing with unforeseen events or even errors, probably induced by their nondeterministic operational environment. More so, specific assumptions made during the development process, e. g., such as predetermined fault models, may prove to be inadequate in a real-world setting.

The *runtime reflection* project presented in this paper addresses these issues by putting forth a combined framework whose constituents (1) allow the *specification* of system properties including detailed real-time requirements, (2) facilitate their *validation and verification* in test and standard systems operation, as well as (3) provide means for performing a detailed on-line *diagnosis* given the occurrence of a system failure.

Basically, the framework comprises two novel approaches, first for dynamically detecting failures in a distributed system, and then secondly for analysing their causes requiring only a minimal communication overhead on the network; in fact, only linear with respect to the number of used monitor components (and only in case of an occurred system error). Unlike failure detection by means of system monitoring, the identification of failures is only performed using a dedicated system's diagnosis if and only if, prior, a monitor has noticed an aberration. As such, there exists no continuous computation and communication penalty for the systems diagnosis, in case the system under scrutiny does work as expected. That is, the diagnostic layer is triggered on-demand.

The runtime reflection framework can be employed during testing or in normal systems operation in the actual product. The monitors used to detect aberrations are executed in parallel with the distributed system, and may later be deployed in terms of "watchdogs" to monitor, say, safety-critical components. While the monitors operate locally on the network with scope on a certain software component, diagnosis—if triggered—obtains a holistic system view to differentiate from the symptoms of a failure, e. g., a missed bus signal, and its actual cause, e. g., deadlock in a remote software component. As such, diagnosis allows—given a set of observations—to deduce the according explanations which then contain the set of conflicts responsible for an aberration.

1.1. Related work

Diagnosis in the automotive domain is typically restricted to either the off-line service and maintenance phase, or if performed dynamically, mostly centred around low-level signal processing, physical process or hardware failures [9, 13, 11], disregarding the complex interactions of distributed functions realised in terms of individual software components, such as will be realised by AUTOSAR-components (cf. [15]). Diagnosis of software-failure is then often reduced to a mere recording of symptoms on the ECU on which a "faulty" component is deployed, which makes it very difficult in practice to reconstruct the exact conditions in which an error originally occurred. To address this problem, various approaches have been realised, for instance, adding additional information about the system under scrutiny in terms of static lookup-tables comprising certain causes and symptoms, reflecting the effects of failures on the system (cf. [12, 10]). Such tables may be obtained prior from a dedicated hazard and risk analysis, FMEA/FTA analysis, or directly from the engineers who designed the system and know about its anticipated ways of failure. Although practically useful, the downside of these solutions is that such knowledge basically constitutes design assumptions, and as such these may be invalidated by the real-world, e. g., when situations occur that cannot be explained using such knowledge.

2. Architectural overview

In this section we give a brief architectural overview over our runtime reflection framework. First, we consider its structure merely in terms of its layers, and without regarding in particular the distribution of its underlying components within the individual layers. Then, we describe the organisation of the individual components of our framework by means of giving a brief intuitive example, reflecting more on the distributed nature and the functionality of our architecture, and the application to be analysed.

2.1. Framework layers

The main architecture is a layered and modular one, in that it supports a separation of concerns; that is, the different tasks of the analysis are handled by separate entities which communicate only through minimal interfaces, as is indicated in Fig. 1.

Let the application under scrutiny be a distributed reactive system consisting of individual components, instrumented and/or annotated to produce an outside-visible stream of (internal) system events.

2.1.1 Logging—Recording of system events

A dedicated *logging layer* in our architecture is the only part of the runtime reflection framework directly known to the (instrumented) application itself. The distributed system, embedded into our framework, employs custom code annotations in order to produce the visible system events, which are then collected and communicated further by our logging layer. The annotations are the only prerequisites, necessary within the application component's source code, in order to be able to use the runtime reflection framework.

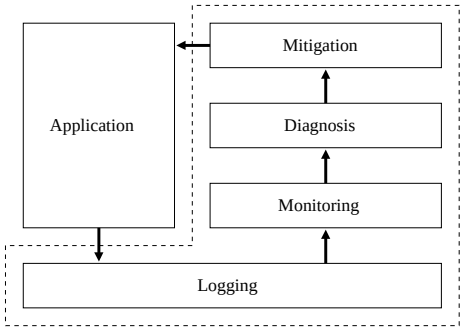

Figure 1. Layers and information flow of the runtime reflection framework.

The logging layer allows to register so-called *loggers*, i. e., software components for observing the stream of system events, and thus, to reflect upon the dynamic behaviour of the executed application. A logger might be part of the application itself, e. g., to extract more general statistics on the overall system utilisation, or to record system events merely to a file during a unit-test session. However, when we employ the logging layer in conjunction with the complete runtime reflection framework, we use the layer to deliberately decouple the distributed application's code from the remaining layers in the framework.

In particular, the application's code does not contain any knowledge on the safety properties which are monitored, and which are then used subsequently for deducing a diagnosis in case of an error. Therefore, we can change the monitored properties and the system description (as used by the diagnosis) even on-the-fly, during their execution without interrupting the running application!

2.1.2 Monitoring—Failure detection

The *monitoring layer* consists of a number of monitors (complying to the logger interface of the logging layer) which observe the stream of system events provided by the logging layer. Its task is to detect the presence of failures in the system without actually affecting its behaviour. It is implemented via *automatically generated monitors* which—each locally with respect to a certain subsystem or system's component—monitor *safety properties* (see Sec. 3).

Intuitively a safety property asserts that "nothing bad happens". Therefore, safety properties impose minimal requirements upon the system which must hold in order to have some sort of a well-defined behaviour. They do not, however, impose a specific behaviour on the system as such. A typical example is the exclusion of certain critical system

states, e. g., one always wants to ensure that in a present state $\neg(gear\_upshifting \wedge gear\_downshifting)$ holds.

If a violation of a safety property is detected in some part of the system, the generated monitors will respond with an alarm signal for subsequent diagnosis.

2.1.3 Diagnosis—Failure identification

We deliberately separate the identification of causes from the detection of failures in terms of a dedicated diagnosis system. The *diagnosis layer* collects the verdicts of the distributed monitors and deduces an explanation for the current system state.

For this purpose, the diagnosis layer infers a minimal set of system components, which must be assumed faulty in order to explain the currently observed system state. The procedure is solely based upon the results of the monitors, and as such, the diagnostic layer is not directly communicating with the application, but rather creates with each diagnosis a "snapshot" of the system at a given time. This bears a major advantage in that no extra messages need to be exchanged between all the monitors in order to obtain a holistic system view.

Our diagnostic layer then infers a system model which incorporates and reflects the observed failures, and compares it with an internal reference model. The differences found constitute possible causes for failure. Basically, this approach is based upon an efficient realisation of the theory of consistency-based diagnosis (see Sec. 4).

2.1.4 Mitigation—Failure isolation

The results of the system's diagnosis can then be used in order to *isolate* the failure, if possible. However, depending on the diagnosis and the occurred failure, it may not always be possible to re-establish a determined system behaviour. Hence, in some situations, e. g., occurrence of fatal errors, a recovery system may merely be able to store detailed diagnosis information for off-line treatment.

In the following sections, for brevity, we therefore focus on the first two layers, monitoring and diagnosis, and establish the methodological foundations for our framework, and sketch its implementation along with some explanatory examples.

2.2. Framework functionality

The logging layer and the monitoring layer consist both of a number of different software components, which are distributed throughout the system under scrutiny; that is, depending on the granularity and number of the system's components. Each local monitor then computes a verdict on the locally observed event stream and provides this verdict for further, subsequent diagnosis regarding the system's general status. The diagnosis and mitigation layers, in contrast to logging and monitoring, are realised in terms of centralised components, which collect the information of the monitors in order to compute and react upon a global system view.

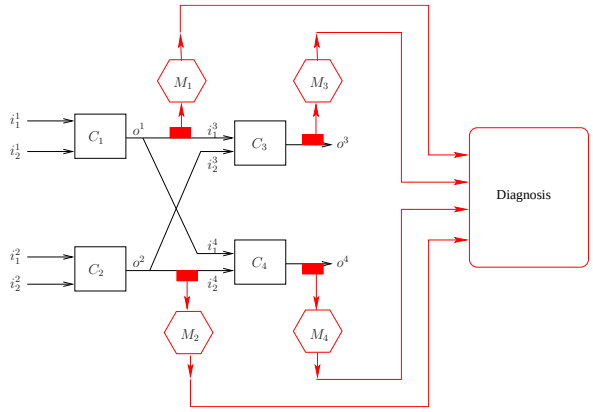

Figure 2. Monitoring a distributed application.

For instance, consider Fig. 2, where we show an example application consisting of four distributed components, $C_1, \ldots, C_4$. To monitor the overall system behaviour of this application, we need to add at least four dedicated monitoring components, $M_1, \ldots, M_4$, to the actual system. Each monitor, M_i, is then locally observing the output of a single component, C_i, and computes its verdict on the correctness of the observed output stream so far. These distributed verdicts are then transmitted back to the central diagnosis component for further treatment.

3. Runtime verification: SALT

In our setting, monitors used for failure detection are automatically generated from high-level specifications. Therefore, we have developed the temporal specifications language SALT [3]. Using SALT, one can define the expected behaviour of the underlying system in a precise manner.

To be conveniently used by engineers, SALT is designed to have the look and feel of the programming language C/Java. The main difference to a programming language is that only assertions on the behaviour of the system are formulated. Thus, the developer does not have deal with actually writing the monitor code but just has to specify the monitor's behaviour.

SALT is similar to the property specification language *PSL* [6] that is used in chip development for expressing functional assertions. In contrast to PSL, however, SALT offers operators for expressing real-time requirements, which is essential for being applicable in the automotive domain.

Unlike other specification languages SALT has a precisely defined syntax and semantics. Its root goes back to Linear Temporal Logic, which is well-studied in the literature.

Furthermore, efficient monitor code checking the assertions formulated in SALT at run-time can be generated, as

explained in [1] in detail.

The results of monitors then constitute the basis for the diagnosis as described in the next section.

Instead of giving a formal syntax and semantics of SALT, let us discuss typical examples describing its expressiveness.

Assume that the value of the $crank\_speed$-signal to observe should always be greater than 0. Using SALT, one writes:

```
always crank_speed>0
```

We can then define an operating mode $Cranking$ by

```
Cranking := always crank_speed>0
```

Now, assume that we have a signal $vehicle\_speed$ and $vehicle\_speed\_old$, we consider their values only meaningful, if their difference is not greater than some δ, since the velocity of a vehicle is limited. We can write

```
VS := |vehicle_speed -
            vehicle_speed_old| < δ
VSpeed := always (not VS implies
                          next VS)
```

to say that at least every second speed signal is valid.

Previously defined assertions can be combined in a Boolean manner:

```
Safe := Cranking and VSpeed
```

Further examples, which are self-explanatory, are

```
always (gas_pedal > 0 implies
        eventually rpm_tracking)

never gas_pedal < 0
```

SALT does further allow the definition of functions, featuring the reuse of specifications.

```
fun Event_To_Mode (event, mode) {
  always (event eventually mode) }
```

SALT provides sophisticated operators to easily support exceptions. For example, we can write

```
(idle_speed > 0 until rpm_tracking)
                    accept_on key_off
```

to say that the idle speed of the engine is positive until switching to operating mode $rpm\_tracking$—unless the driver switched off the car.

SALT provides a notion of time which can be used to formulate real-time requirements. While in SALT an abstract notion of time units is used, a unit can be related to a real-time value (e.g., one unit equals 10ms) during the deployment phase of monitors.

The assertion that we always get a new value denoting the current $vehicle\_speed$ within the next 10 time units can be formulated as

```
always (next(vehicle_speed) in [0,10])
```

The operation of the catalytic converter is controlled by values delivered by the lambda probe. These values might be ignored, if they are outdated. In the testing phase, we log whenever a value is ignored and test by asserting

```
always (ignore_lambda implies
        last(l_correction)  >  200])
```

that indeed only values older than 200 time units are ignored.

In [1], we describe how to generate efficient monitors checking the real-time properties expressed in SALT. However, there is an intrinsic problem when facing this goal: A monitor can have at most a *finite* view on the system's behaviour over time, whereas temporal assertions are usually defined over infinite behavioural traces. This leads to complications as can be seen by the following example:

Consider the assertion

```
(not ACC_init) until idle_mode
```

stating that the adaptive cruise control may not be initialised before the car finished the start-up phase.

When ACC_init is called before entering the idle mode, the monitor should clearly report a failure. When the start-up phase is over and the car enters idle mode, the assertion does definitely hold. Actually, the monitor is of no need anymore and could be stopped to save resources. While in the start-up phase and no ACC_init is observed, the assertion is neither wrong nor satisfied: It depends on whether ACC_init or $idle\_mode$ is observed first.

One distinguishing feature of our runtime monitoring framework is its three-valued semantics. In contrast to existing work in this domain, we generate monitors that classify the last case as *inconclusive*, exactly following the practitioners intuition (see [1] for details).

4. Failure diagnosis

Diagnosis in the runtime reflection framework is used to separate observed symptoms of a failure from actual causes. Essentially, the diagnosis engine, if triggered, uses the inputs of the monitor components monitoring only local components of a distributed system, and deduces possible explanations for observed aberrations. This way, the framework accommodates the complex interplay of distributed software functions and avoids so-called false-negatives when trying to analyse a failure.

Basically, diagnosis uses two inputs to analyse the system: a) a propositional *system description* comprising all components of a distributed system and their causality, and b) the output from the individual monitors watching over the components. It then tries to infer explanations for matching the monitor's observations with its internal system description.

Since the exact behaviour of a component over time is specified in terms of monitored properties, the system description of the diagnosis engine does not contain any behavioural models. More so, due to the combination of runtime verification by means of monitors and model-based diagnosis, our engine infers possible explanations for an observed failure based on the satisfaction of a propositional logic model [2].

Example. Consider again Fig. 2. One possibility of diagnosing such a system of four constituents is to use the following simple system description SD:

$$SD = \left\{ \begin{array}{l} ok(i_1^1) \wedge ok(i_2^1) \wedge \neg AB(C_1) \Rightarrow ok(o^1), \\ ok(i_1^2) \wedge ok(i_2^2) \wedge \neg AB(C_2) \Rightarrow ok(o^2), \\ ok(o^1) \wedge ok(o^2) \wedge \neg AB(C_3) \Rightarrow ok(o^3), \\ ok(o^1) \wedge ok(o^2) \wedge \neg AB(C_4) \Rightarrow ok(o^4) \end{array} \right\},$$

where ok is the predicate denoting the that the corresponding input or output value is correct, which is determined by the individual monitors in the system. The AB predicates in turn denote that a component is "abnormal", i. e., not working as expected. The causality is then given by the propositions occurring in more than one propositional sentence.

Notice the semantics of what ok actually encodes is entirely up to the property checked by the according monitor.

If a failure occurs, e. g., $\neg ok(o^3)$ is observed, the diagnosis engine then infers all possible configurations where $\neg ok(o^3)$ holds, such that the overall system description, the observations, and the set of AB-literals are consistent. In this example, we obtain the following abstract conflicts explaining the observations:

$$CONF = \left\{ \begin{array}{l} \{C1, C2, C3, \neg C4\}, \\ \{C1, C2, \neg C3, \neg C4\}, \\ \{C1, \neg C2, C3, \neg C4\}, \\ \{C1, \neg C2, \neg C3, \neg C4\}, \\ \{\neg C1, C2, C3, \neg A2\}, \\ \{\neg C1, C2, \neg C3, \neg C4\}, \\ \{\neg C1, \neg C2, \boxed{\text{C3}}, \neg C4\} \end{array} \right\}.$$

The last solution seems the most obvious one, since it assumes that only one component failed: only C_3 is marked faulty in order to explain why $\neg o3$ holds. But we cannot be sure of that, since the values of $o1$ and $o2$ are not modelled; hence, the other six solutions.

Formally, our diagnosis approach is based upon the theory of model-based diagnosis [14, 8]. However, unlike in the original theories, which use first-order models comprising causality as well as behaviour, our approach is strictly reduced to propositional models and causality, since behavioural accordance is more efficiently encoded using the automatically generated monitors from our specification language SALT. As such we provide efficient means of solving the diagnosis problem, and if desired, also on the fly when the failures occur; that is, at runtime. Benchmarks of our diagnosis engine are presented in [2].

5. Example

Let us put the runtime reflection framework to use by illustrating a simple example taken from vehicle dynamics. In Fig. 3, a distributed system is exemplified which, depending on some horizontal force, compensates the caused horizontal drift by actively influencing the car's steering. Such compensation may be necessary due to very strong winds coming from either side of the vehicle while driving.

Since this setup resembles a highly safety-critical system, the components, C_I and C_R, which implement the main control algorithm for generating an according PWM-signal are laid out redundantly (possibly) on two ECUs. Amongst other values they process the vehicle's speed, and a sensor value, *hforce*, encoding the horizontal force. Basically, C_I contains an algorithm based on integer values, whereas C_R resembles the actual control algorithm based on real or float values, such that the output of C_R can be checked against C_I and aberrations are detected.

Using the framework's means of specification and monitoring, we can then specify four monitors which check whether the necessary signals for control are present in a given time-frame in milliseconds (see M_1 and M_2), and secondly, whether the respective changes between the PWM-signals that actually affect the steering are not invalid. For example, some abrupt change, i. e., values greater than a predefined δ, in the signal's values may result in a too large steering angle and destabilise the vehicle.

In case of an aberration, the monitors' results are propagated to the diagnosis engine for further analysis. It is then able to differentiate whether the cause of the failure originates in the vehicle dynamics system itself, e. g., in one of the components determining the PWM-signals, or whether the fault was induced, for instance, elsewhere from the environment. Depending on the results, it may be necessary to shut-down the system and notify the driver of the car.

6. Conclusions

The presented framework for failure analysis provides tools and methods that enable component-based distributed embedded systems, such as automotive systems, to reflect upon their overall system status at runtime. The framework's application is flexible, in that it can be used either in testing the system during development, or be deployed with the actual product to analyse safety-critical parts at runtime.

Due to the layered architecture and the efficient combination and realisation of different techniques for reasoning about distributed embedded systems, we eliminate computational overhead for diagnosis. That is, our component-oriented approach triggers diagnosis specifically at the occurrence of a failure, which avoids a continuous computational effort. Additionally, the use of independent and local monitors in order to observe specific components, avoids an expensive communication penalty in that no extra diagnostic messages need to be exchanged between the respective

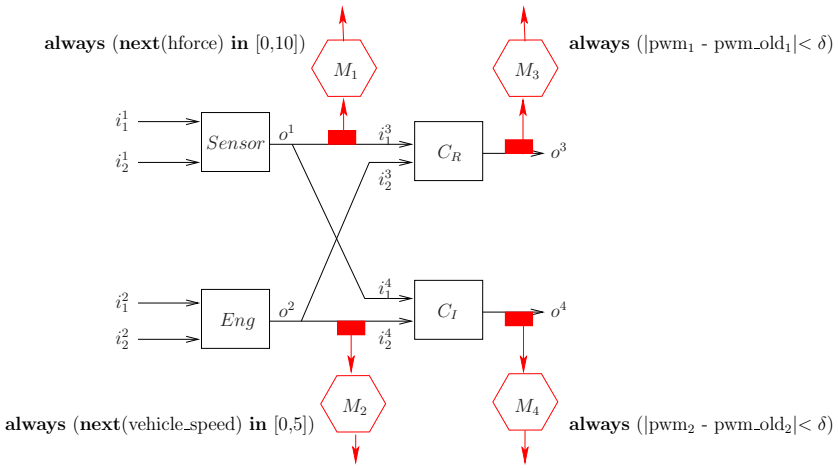

Figure 3. Monitoring a vehicle dynamics system.

monitors in order to come to a verdict regarding a system's overall status.

We have successfully implemented the ideas presented in this paper (see `http://runtime.in.tum.de/`), and are currently in the process of streamlining the entire architecture for ease of integration and further extensibility towards recovery measures. The latter were, on purpose, not intensively dealt with in this paper, since they constitute highly domain-specific knowledge and methods, which are not necessarily applicable to all real-time or reactive systems alike. Consider, for instance, the differences between distributed control systems, and the wide area of the "Infotainment" domain.

Finally, the approach is based upon the notion of software-components, making it suitable for future deployment in middleware-based environments, such as presented by the AUTOSAR infrastructure. The monitor components, the loggers, the diagnoser, as well as the system's components can be realised in terms of AUTOSAR components communicating over dedicated ports and a common API. Furthermore, our approach complements, existing model-based approaches to automotive software development in that we offer means to realise abstract requirements using a formal specification language that can be used to automatically generate the monitor components.

References

[1] O. Arafat, A. Bauer, M. Leucker, and C. Schallhart. Runtime verification revisited. Technical Report TUM-I0518, Technische Universität München, 2005.

[2] A. Bauer. Simplifying diagnosis using LSAT: a propositional approach to reasoning from first principles. In *Proc. CP-AI-OR*, volume 3524 of *LNCS*, Prague, Czech Republic, June 2005. Springer-Verlag.

[3] A. Bauer, M. Leucker, and J. Streit. Salt—structured assertion language for temporal logic. Technical Report TUM-I0604, Institut für Informatik, Technische Universität München, Mar. 2006.

[4] A. Bauer, J. Romberg, and B. Schätz. Integrierte Entwicklung von Automotive-Software mit AutoFOCUS. In *Proceedings of the 2. Workshop Automotive Software Engineering*, Ulm, Germany, Sept. 2004. Gesellschaft für Informatik.

[5] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005.

[6] B. Cohen, S. Venkataramanan, and A. Kumari. *Using PSL/Sugar for Formal and Dynamic Verification*. VhdlCohen Publishing, 2 edition, 2004.

[7] M. Conrad. Systematic testing of embedded automotive software—the classification-tree method for embedded systems (CTM/ES). In *Perspectives of Model-Based Testing*, number 4371 in Dagstuhl Proc. Schloss Dagstuhl, 2005.

[8] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *AI*, 32(1):97–130, 1987.

[9] R. Isermann, editor. *Überwachung und Fehlerdiagnose — Moderne Methoden und ihre Anwendungen bei technischen Systemen*. VDI-Verlag, Düsseldorf, 1994.

[10] R. Isermann. Model-based fault detection and diagnosis: status and applications. In *Proceedings of the 16th IFAC Symposium on Automatic Control in Aerospace*, St. Petersburg, Russia, June 2004.

[11] E.-J. Manders, G. Biswas, P. J. Mosterman, L. A. Barford, and R. J. Barnett. Signal interpretation for monitoring and diagnosis, a cooling system testbed. *IEEE Trans. Instr. and Measurem.*, 49(3):503–508, June 2000.

[12] M. Nyberg. *Model Based Fault Diagnosis: Methods, Theory, and Automotive Engine Applications*. PhD thesis, Linköpings Universitet, June 1999.

[13] M. Nyberg, T. Stutte, and V. Wilhelmi. Model based diagnosis of the air path of an automotive diesel engine. In *IFAC Workshop: Advances in Automotive Control*, Karlsruhe, Germany, 2001.

[14] R. Reiter. A theory of diagnosis from first principles. *AI*, 32(1):57–95, 1987.

[15] C. Salzmann, M. Thiede, and M. Völter. Model-based middleware for embedded systems. In P. Dadam and M. Reichert, editors, *GI Jahrestagung (2)*, volume 51 of *LNI*, pages 3–7. GI, 2004.

[16] P. Thoma. Automotive electronics—a challenge for systems engineering. In *Design, Automation and Test in Europe (DATE '99)*, page 4, 1999.