Model based dependability evaluation for automotive control functions

Saša Vulinovic [1], Bernd-Holger Schlingloff [2,3]

1: Institut für Technische Informatik und Mikroelektronik, Technische Universität Berlin, 10587 Berlin, Germany
sasa@vulinovic.de

2: Fraunhofer Institut für Rechnerarchitektur und Softwaretechnik FIRST, Kekulèstr. 7, 12489 Berlin, Germany
holger.schlingloff@first.fraunhofer.de

3: Institut für Informatik, Humboldt-Universität zu Berlin, Rudower Chaussee 25, 12489 Berlin, Germany
hs@informatik.hu-berlin.de

Abstract

In this paper, we study the evaluation of reliability for embedded functions in automotive software. With the decrease of circuit size the probability of hardware failures within a chip increases. In order to assess fault tolerant designs for automotive software it is essential to be able to predict the failure rate of the realized function. We present a model based fault injection technique, where faults are injected according to a distribution of probabilities in the function model. Using the MATLAB/SimuLink® tool chain, we are able to simulate the faulty behavior of a function even if the realization is distributed amongst several embedded processorsr. The results allow assessing the dependability of certain safety critical functions in the overall system design.

Keywords: Model simulation, fault injection, multi-level model based testing, verification plan, SWIFI, automotive ECU, Soft-CPUs.

1. Introduction

Today, automotive embedded control units (ECUs) are increasingly designed using model-based methods. Starting from informal requirements, a graphical model is constructed, which then is refined in various ways.

The first step in the design is the construction of a platform independent model (PIM) which reflects the original requirements. The PIM is transformed into a platform specific model (PSM) by stepwise refining and implementing parts of the original model. Finally, a code generator can automatically produce target-specific G-code from the model. Other downstream tools include software- and hardware-in-the-loop (SIL and HIL) test environments as well as rapid prototyping and onboard testing solutions.

A major problem in automotive software engineering today is the number and variety of ECUs within a vehicle. Moreover, software functionalities are increasingly interwoven with each other, yielding an additional level of design complexity.

A possible solution for these problems is to use an interconnected network of multifunctional and standardized ECUs. For such a solution, in the early design stages functionalities must be developed in a processor independent manner. Later, the sub-functions and their implementations are distributed within the network. This approach simplifies the variety and the number of necessary ECUs. However, due to the high interdependency, failures can spread across the whole system and can have unforeseen consequences for the behavior of the vehicle.

In this paper, we consider failures of safety-critical systems that arise from random faults in the hardware or execution of the software. That is, we are not dealing with systematic faults in the design of the system, but with exceptional behavior which origins from anomalies in its operating environment. We present software implemented fault injection techniques to simulate the effects of errors on different abstraction levels. This way we are able to determine the behavior of the system in the presence of faults and to identify design weaknesses in order to improve the overall dependability.

This paper is organized as follows. In section 2, we define the process of software implemented fault injection. Subsequently, we describe which types of errors can be injected at the various levels in the design process. In section 4, we discuss the need and realization of co-simulation between different modeling levels. Finally, we conceive an integrated fault injection tool environment and summarize our results.

2. Software Implemented Fault Injection

The reliability of embedded safety-critical systems can be assessed by *fault injection*: the system is run in normal operating mode while selected faults are deliberately imposed onto the system and its behavior is measured. In this way, the robustness of the system and its ability to prevent negative consequences of random faults can be determined.

One possibility to realize this approach is by *software implemented fault injection* (SWIFI). Here, the system under test (SUT) is exposed to faults, which are simulated by software and injected via a CPU debugger or by additional low-level code. In section 3, we describe how this can be done in a model-based software development process. The faults are carefully selected to cover systematically all possible malfunctions that could occur.

There are two major advantages of software implemented fault injection as compared to approaches that utilize physical fault injection. Firstly, the method is nondestructive, and reproducible results are obtained at minimal cost. Secondly, the method can be tuned such that it produces statistically significant results which can be used in a certification and validation process. However, since it requires additional calculations, the injection process can alter the performance or timing of the SUT.

3. Model Levels and Associated Faults

According to the level of abstraction, SWIFI makes it possible to analyze the effects of faults onto the various artifacts in the model-based design flow. In such a process, the target is developed via several stages at decreasing levels of abstraction. Each level offers specific degrees of freedom and therefore different possibilities for faults to be injected. We distinguish the following four levels.

- Functional level
- Block level
- Code level
- Gate level

3.1 Functional Level

The functional level consists of abstract blocks (e.g. ACC (adaptive cruise control), ABC (active body control), digital ignition system, or airbag control system) and models the interconnections between these blocks. Commercial simulation tools allow the test engineer to mark individual components as faulty, run the simulation and monitor the effects on other functions. Basic faults can be defined by specifying data domains and their associated ranges together with faulty behavior on blocks, sensors or actuators. The results of these simulations can be used to build a complete fault-tree for an FMEA.

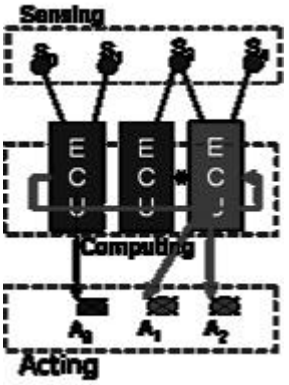

Figure 1: A network of ECUs and peripherals

This figure shows an example of the interconnection between different sensors, actuators and embedded control units, where the realized functions are distributed between the various components. With suitable tool support, it is possible to simulate e.g. the effects of a sensor failure.

In an ongoing work, we propose to combine function-level fault injection with formal specification methods. We give a formula in some temporal logic which reflects the correctness conditions to be checked. Then, random faults are injected into the functional model as described above. We propose to use model checking techniques to evaluate the satisfaction of the safety requirements in the faulty model. The resulting counterexamples of the model checker can serve for threat analysis in the FMEA.

Moreover, the formal specification can help in the selection of faults to be injected. Model checking a negation of the specification formula on an abstract level gives all combinations of simple faults which lead to the violation of safety. These combinations can be used in a simulation on the more concrete level to assess the associated risk.

3.2 Block Level

The block level represents the algorithmic realization of the functions. In a block-level model, the abstract macro blocks are present as concrete implementations of the functions. Both a platform independent and a platform dependent model consist of blocks, where the latter is enhanced with allocation of blocks to the given hardware topology. Basic blocks like constants, conditional decisions, or simple logic functions may be nested to form algorithms that are more complex.

There are two classes of possible faults for injection on the block level: *Mutants* and *saboteurs*. A mutant is a distorted block which replaces an original block with a slightly modified functionality. A sabo-

teur disturbs the original communication between (basic) blocks by some additional functionality.

In accordance with [10] we distinguish between the following classes of mutants

- **Mixing Lines**
  Permutation of connections between basic blocks

- **Random Selection**
  Substitution of a basic block with another basic block, guided by a substitution list

Saboteurs are selected from the following classes:

- **Break Line**
  Cutting of a connection between two basic blocks. The emitting block is terminated with a sink element, and the receiving block is connected to a constant.
- **Gain/Offset**
  Amplification of the original signal or adding of a static offset
- **Noise/Chirp**
  Adding of a dynamic signal to an original signal. The additional signal ranges from periodic functions to random noise, defined in tables or arithmetic functions.

In a large model, there are numerous possibilities for mutants and saboteurs to be injected. Moreover, there is an astronomically large number of combinations of such faults. Therefore, it is essential to establish a realistic fault distribution model. Here "realistic" means that the fault distribution model guarantees that the chosen faults reflect actual faults, which are likely to occur in an average operational usage.

To be applicable for a quantitative estimation of the reliability of the system, the selection of faults has to be statistically independent. Fault distribution models can be designed from experimental data, or from data extracted from error memories in customer vehicles. Such a fault model can also be used for generation of test cases or the evaluation of other verification tools.

3.3 Code Level

An important advantage in model-based software design is the possibility of automatically generating target-specific C-code. Code generation allows maintaining design consistency through the various modeling levels. A modification on an abstract level causes the corresponding changes on the more concrete level at the push of a button. Existing code generators transform basic blocks to highly optimized code, which is comparable in efficiency to manually written code. However, the optimization complicates the tracing of faults on the target to the model.

To induce erroneous behavior on the instruction level we use the approach we proposed in [11]:

The instruction level is modeled as sequence of instructions. Each instruction consists of three major parts:

- Operands
- Pointer to operands
- Functionality

We take the single bit-flip model as basis for all modifications of the instruction parts. The programmed functionality is mutated by flipping a single bit in the machine instruction word. Operands and pointers are mutated by inducing a single bit flip on CPU registers or memory locations. By applying these mutations to all instructions of the application, a fault dictionary is built.

By stimulating the application with realistic loads, the execution flow is recorded and profiled. The profiling data provides the weights for creating a list of faults to be injected. Every entry in the fault list contains:

- the location of the fault to be injected
- a particular point in time for fault activation
- a fault type, that is feasible for this particular instruction, and
- an application load

The fault list is read and transformed to create executable debugger scripts. The debugger applies the scripts; the system behavior and its output is recorded in a fault effect data base. This data base is evaluated off-line to determine the simulation results.

3.4 Gate Level

The concrete hardware of the system is analyzed at the register-transfer level. Fault injection can help to assess the reliability of certain integrated circuits. The system can be seen as consisting of switching elements and memory cells or memory contents. Thus, faults concern latches (bi-stables) and logic gates. The faults are injected by modification of memory cell contents and gate functionalities in a HDL-description of the circuit. In order to be able to do so, it is necessary that such a model is available. For many commercially-off-the-shelf processors, such a description is not available since it is a core intellectual property of the chip manufacturer. However, for certain CPUs such as the LEON2 processor [18] the gate level design is freely available.

The number of possible faults to be injected is on the gate level even larger than on the levels above. Therefore, selection of faults is done by random choice or according to some assumptions on the spatial distribution of switching elements. The fault injection is done by modification of the processor description and

execution of the modified code in a HDL simulation tool.

The possible faults in a gate-level fault injection are strongly dependent on the hardware technology of the integrated circuit. Faults in switching elements are only relevant for technologies below 65 nm; thus for present-day technology only memory errors have to be considered. In CMOS circuits, the majority of faults consists of single-bit-flips; see next section.

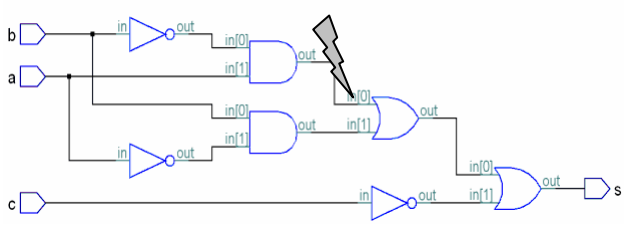

Figure 2: Faults on the level of gates

4. Co-Simulation

Today most innovations in the automotive sector are realized by software-intensive systems. New functionalities require more and more computing power. System designers must achieve more performance, but with less power consumption and fewer cabling, while still maintaining reliability and availability. These goals can be reached by integrating more than one function on each ECU. To this end, each ECU must be equipped with a high performance CPU or even with a multi-core CPU. Integrated circuits with 400 million of transistors and more are possible, especially with the emerging 65nm technologies. We expect that the reasonable low-price per MIPS will also be used for spare on-chip blocks, to enhance the reliability of ECUs and to reduce maintenance cost.

Another possibility is the use of configurable devices (e.g. FPGAs) in ECUs. This could solve several major problems at once:

- Obsolescence of CPUs and MCUs: Soft-cores can be used with almost any FPGA.
- Configurable interfacing with standardized or custom busses: Interfaces for most busses can be easily integrated and replicated by demand.
- Field upgradeability: Hardware updates which improve reliability or performance can be done as easily as software updates.

For the current mainstream semiconductor technologies at 130nm and larger, it is sufficient to inject faults at instruction level to imitate the Single Event Upset Event in a CPU core. A study [14] has shown that 95% of all faults at the gate level can be modeled by inducing a single bit flip at the instruction level. This holds true for technologies where the vast majority of faults happens in a flip-flop. However, for the newer technologies studies like [15, 16] showed that logic gates are susceptible to errors at the same order of magnitude as the flip-flops. This results in multi bit flips in the memory elements and has to be considered.

These low-level faults can no longer be injected on a MATLAB/Simulink® basis. Therefore we need an additional HDL- simulator (e.g. ModelSim® [17]), which is able to handle this modeling level. By applying a fault model to the HDL-Code of the CPU we are able imitate the CPU function in presence of low-level faults.

The realization of a fault injection at register-transfer or gate level can be easily done. Consequently, simulation time will rapidly increase.

4.1 Speeding-up Fault Injection Campaigns

To address the problem of fatiguing simulation times we are currently investigating the relationship between different modeling levels and implementing some improvements in the fault selection process. This will enable us to estimate the reliability on a lower level by simulating at a more abstract level.

We start with the implemented model as described above. Is has to contain enough details to be able to generate the fully working optimized C- source code. The generated code should be annotated for the fault injection process. This allows defining different failure rates for each sub-block.

Presently we are developing a tool for fault injection on code and gate level. We choose a couple of software applications and categorize the system outputs in presence of faults as shown in the following figure.

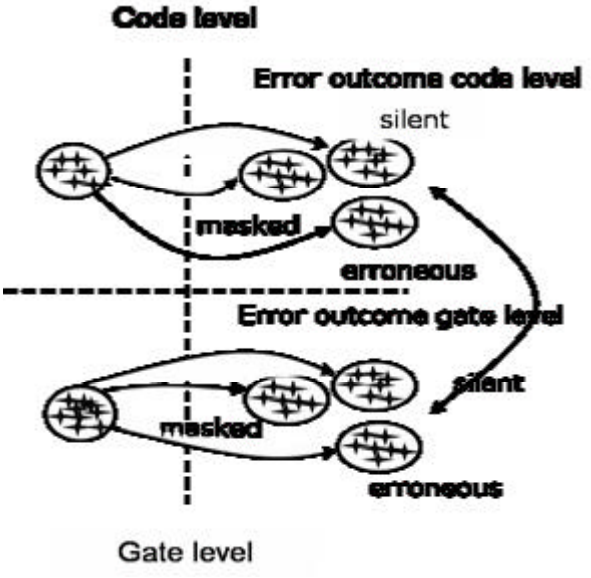

Figure 3: Classification of induced malfunctions

A fault selector module chooses the faults, which will be injected by a fault injector module and logged in the fault effect databases. This will be applied for each level separately. In order to asses the quality of the

fault selector module the fault effects are classified into these categories:

- silent failure: fault is not activated yet, remains latently present, not masked
- erroneous failure: fault was activated and showed deviation from the reference
- masked failure: fault was activated, but got masked

At instruction level, the procedure is conducted as described in the previous section 3. For obtaining results at the gate level, the capabilities of the HDL-Simulator are used to implement mutations and saboteurs for the gate level elements, while executing the applications on a HDL soft core CPU (e.g. LEON2 [18], ARM). Again, it is important to keep in mind which technology will be used for the real system. For example, the 65nm technologies have to be simulated with faults that are chosen equally from gates and flip flops to obtain a realistic fault behavior [19, 20]. A comparison between the different fault classes for both levels generates results in the demanded relationship.

The key element to conducting a fast and accurate simulation is the fault selector. We could speed up the whole system simulation if it is possible to identify faults for which the error result is already known. To decide whether the error is to be injected we evaluate the gate net list and consider the instruction that shall be executed. An error will be masked if some other input of the selected gate is sufficient for keeping the fault from spreading to the following gates. Figure 2 shows an example. The output of the or-gate is always "1", because the other input is known to be "1" also. Hence any potential fault induced by the other input has no effect and does not need to be simulated. Similar rules can be defined for each basic gate and combination of gates.

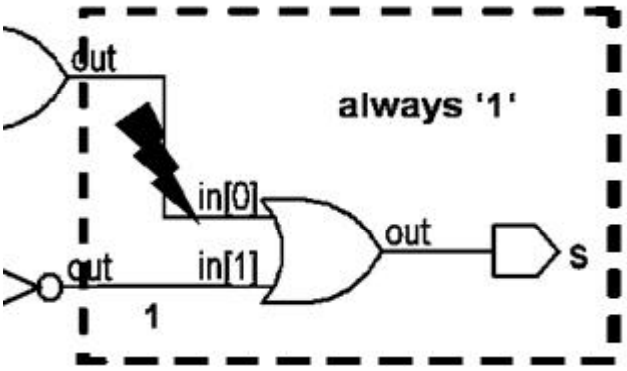

Figure 4: Bit flip on „in[0]" is masked by other input, the result does not change

The logical values for this algorithm are obtained by a statistical approach. For a simple RISC the net list values are sampled for some clock cycles, while executing predetermined test instructions. These include structured tests, with dedicated combinations of instructions with operands. The performing of a cross correlation results in a logic value marked with a probability, which depends on the instruction, the addressing mode and the operands. This has to be done only once for the desired CPU.

A further advantage is given by comparing the results to the "no operation" state of the CPU. Signals that differ from the "no operational" value indicate that the component connected to this signal is active during a particular instruction execution. Faults at gates with a high activity probability need to be simulated, while others can be ignored. Usually the gate activity list is not available from the CPU manufacturer. However, for soft core CPUs gate-level descriptions are available to the customer. This enables us to obtain this data using the method described above.

4.2 Integration into MATLAB/SimuLink®:

Finally we create an abstract model with all the gathered information. A possible realization could be a modification of the internal MATLAB debugger, which imitates a faulty gate level model of the CPU. With an extended examination under different conditions the additional debugger functionality can be parameterized by

- CPU model,
- layout technology,
- operational frequency,
- operating temperature, and
- ageing of the hardware.

In order to calibrate these parameters, however, appropriate data from a large number of measurements is needed. At present, these data are not available; thus, reasonable assumptions on the influence of the parameters must be made.

5. Tool Environment

In this final section, we discuss the conception of an integrated tool environment for model based software implemented fault injection. This method shows different results on different levels. For a particular assessment, it is important to choose the appropriate level of abstraction. On the functional level, it is possible to investigate the linkage of failures of macro blocks, whereas the robustness of the utilized algorithms is assessed on the implementation block level. The effects of random hardware failures can be modeled at the instruction or gate level. Ideally, an integrated tool should support all of these levels to an appropriate level of detail.

For an encompassing overview of the system reliability it is important to be able to plan in advance which entities should be assessed and how this should be done.

This planning process should also be supported by an integrated tool.

On the abstract level, the top-level behaviors can be identified. The supporting system should automatically extract domains and co-domains from the design and should offer them for fault injection. The supporting system could automatically specify an interface test, following the guidelines given by the tester.

On the implementation level the tester should be able to define "hot spots" where he can identify the most important points in the design and adjust the "simulation depth", ranging from simple number of test cases to marking of components for formal verification.

The defined test cases must be kept in an appropriate database. They are needed by an evaluator module, which constructs an executable test plan at the push of a button.

A report generator module could complete the framework. It would document the effort of the test plans by providing exact information about the specified and generated test results. To be more expressive coverage metrics could be calculated and included:

- **Structural coverage**: Shows which specified modules, blocks, and functions are tested. Common best practice coverage metrics are assigned to standard library blocks.
- **Data coverage**: Shows the amount of tested input values compared to the specified (co-) domains and the coverage of boundary cases
- **Code coverage**: Shows which part of the G-code is left untested
- **User-defined coverage metrics**

In summary, the complete framework could assist the systems designer to determine which parts of the application should be simulated in which way. The framework could also support the systems engineer by automatically extracting structural information from the model and offering him the possibility to specify all relevant simulation constraints. The framework then would run the simulations, inject faults on different levels and log the system behavior in a fault effect database. A report generator would document the specifications, the injected faults and their consequences. These results would be accompanied by appropriate coverage metrics in order to confirm their statistical relevance for the quantitative assessment of reliability.

6. Conclusion and Further Work

In this paper we presented model-based software implemented fault injection approaches for the reliability evaluation of automotive embedded control units. The methods can be applied on different modeling levels throughout the design cycle, yielding qualitative different results on each stage.

We also described in detail the implementation of the code-level fault injection augmented by a gate-level co-simulation, and gave concepts for an integrated tool environment. We have already begun to implement an academic prototype of such an environment.

More work, however, is needed on some of the issues raised here. The relationship between the faults and their effects within a level and between different levels needs to be further investigated. The influence of parameters such as environmental conditions on the assessment of the system under test has to be established by appropriate research and measurements.

New technologies and the increasing demand for computational performance require the development of new abstract models which need to be integrated in the existing design tool chain.

Finally, the relation between quantitative and qualitative assessments is yet to be researched. Currently, we are implementing some of the described methods and apply them on an embedded control system with a LEON2 CPU in a VIRTEX2 FPGA board. The results of this study will help to refine the used models and enhance the universality of the proposed approach.

7. References

[1] J.-C. Fabre, F. Salles, M. Rodriguez-Moreno, J. Arlat: **Assessment of COTS Microkernels by Fault Injection**; Technical Report, LAAS-CNRS, Toulouse 2001

[2] M.-C. Hsueh, T. Tsai and R. Iyer, **Fault Injection Techniques and Tools,** Computer, vol. 30, pp. 75-82, 1997.

[3] J. Clark, D. Pradhan; Fault Injection: **A method for Validating Computer-System Dependability**, IEEE Computer, June 1995, pp. 47-56

[4] M.C.Calzarossa and S. Tucci (Eds.): **Measurement-Based Analysis of System Dependability Using Fault Injection and Field Failure Data**. Performance 2002, LNCS 2459, pp.290-317, 2002.

[5] M.C.Calzarossa and S. Tucci (Eds.): **Measurement-Based Analysis of System Dependability Using Fault Injection and Field Failure Data**. Performance 2002, LNCS 2459, pp.290-317, 2002.

[6] Rodriguez, M., Salles, F., Fabre, J.C., Arlat, J.: **MAFALDA: Microkernel Assessment by Fault Injection and Design Aid** Proceedings of the 3rd European Dependable Computing Conference (EDCC-3), 143–160, 1999

[7] J. Carreira, H. Madeira and J. G. Silva, **Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers**, IEEE Transaction on Software Engineering, vol. 24, pp. 125-136, 1998.

[8] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, **FERRARI: A Flexible Software-Based Fault and Error Injection System**, IEEE Trans. on Computers, vol. 44, pp. 248-260, 1995.

[9] Potential and Challenges for Model-based Development in the Automotive Industry; in "**Business Briefing: Global Automotive Manufacturing and Technology**", World Market Research Center, Oktober 2000

[10] Capeware Technologies GmbH: **Cape/C**. http://www.capeware.de

[11] F. Geue: **Automatische Fault-Injection in SIMULINK Modelle**; Studienarbeit, Fachhochschule für Technik Esslingen, 2000

[12] S. Vulinovic und H. Schlingloff: **Applikationsgeführte softwareinduzierte Fehlerinjektion eines fehlertoleranten Stellwerkscomputers**; In: 16th ITG/GI/GMM Workshop "Testmethoden und Zuverlässigkeit von Schaltungen und Systemen", FhG I-AS/EAS, Dresden (Feb. 2004).

[13] S.G. Elbaum and J.C. Munson: **Evaluating regression test suites based on their fault exposure capability**; Journal of Software Maintenance: Research and Practice Volume 12, Issue 3 , Pages 171 – 184

[14] P.E. Ammann, P.E. Black, W. Majurski; **Using Model Checking to Generate Test from Specifications**; Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), Brisbane, Australia (December 1998), edited by John Staples, Michael G. Hinchey, and Shaoying Liu, IEEE Computer Society, pages 46-54

[15 Joakim Ohlsson: **On Concurrent Error Detection and Error Propagation**, Dissertation, Department of Computer Engineering, Göteburg 1995

[16] Alfredo Benso, Paolo Prinetto edt., "**Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation**", Kluwer Academic Publishers,

[17] P. Shivakumar, "**Modelling the Effect of Technology Trends on Soft Error Rate of Combinatorical Logic**", Int. Conf on Dep Systems and Networks, June 2002

[18] Mentor Graphics: "**ModelSim"** . http://www.model.com/

[19] Gaisler Research: "**LEON2"-**CPU. http://www.gaisler.com/products/leon2/leon.html

[20] **Assessing the Fault Tolerance of Embedded Software through Application of Machine Instruction Mutations**, Proceedings of the IASTED International Conference on Modelling and Simulation MS'99 (4.-8. Mai 1999, Philadelphia USA), S. 541-549. ISBN 0-88986-247-8

[21] Shekhar Borkar, Tanay Karnik, Vivek De: **Design and Reliability Challenges in Nanometer Technologies**, DAC 2004

[22] Alan Matthews: **Das rollende PLD**, Design&Elektronik „KFZ-Elektronik", Feb. 2005

[23] The MathWorks: "**MATLAB/Simulink**"; http://www.MathWorks.com

A preliminary version of this article has appeared at the WMSCI 9th World Multi-Conference on Systemics, Cybernetics and Informatics, July 10-13, 2005 - Orlando, Florida, USA, © IIIS.