Model-driven development of critical perception components using Simulink

Tino Brade, Sebastian Zug, Jörg Kaiser

Model-driven development of critical perception components using Simulink
(Invited Paper)

Tino Brade, Sebastian Zug, Jörg Kaiser

Department of Distributed Systems (IVS)
Otto-von-Guericke-University Magdeburg
(brade,zug,kaiser)@ivs.cs.uni-magdeburg.de

Abstract. Modern sensor-actuator applications combine a large number of components (sensing devices, processing nodes, networks) and implementing complex interactions between them. Due to failures and other sensor inherent insufficiencies the intended control function is adversely affected. This cannot be tolerated in safety critical applications. In some cases replication and voting may be possible. But this is no general solution. Many sensors cannot be replicated because of cost or their operating principles. This demands other failure detection and handling mechanisms to meet application requirements. Model-driven development techniques can be exploited here to adjust the failure handling to the needs of the application. In this paper, we propose a Simulink framework that supports the entire development chain. This includes a new description technique, design verification using regular expressions in combination with a model generator. In contrast to existing approaches, our scheme applies one modeling concept and one development environment throughout the entire process.

1 Introduction

Electronic devices that are embedded in vehicles like cars or airplanes need to fulfill a varying degree of safety requirements. For cars, safety has emerged to one of the main issues because cars include more and more assistance systems that are taking over control of the car in critical situations. Because these critical situations occur in the physical environment of the car and are perceived by the car's sensor system, the reliability of the computational chain from the raw sensors to the provision of application defined high level environment information is crucial. A similar situation occurs in avioncs when moving from "see and avoid" to "sense and avoid" [1]. When sensors are affected by a failure, the subsequent filters and evaluators may become unstable.

Detecting such failures is therefore a basic need. The problem with detecting sensor failures comes from the nature of sensor data. Sensor failures may change the sensor data is many subtle ways and are called failures in "continuously valued" data [2]. This means that any value may be valid. Schemes that can handle such failures are based on replicating sensors. This however is not always

possible because of costs and also because of sensor characteristics, e.g. active sensors like radar, laser scanners or even simple infrared or ultrasound distance measurement devices cannot be operated by simple replication.

We propose a scheme that provides failure detection on the basis of analyzing the data provided by a sensor. This data-centric approach [3] tries to identify the error in sensor data rather than diagnosing the fault that led to the erroneous data. We provide a number of detectors, each tailored for a specific (potential) sensor failure. Basically, we follow a model driven approach and suggest to model the individual behavior of a sensor under the respective failure conditions. We aim at providing a description that can be evaluated in the design process to generate interfaces for checking compatibility with the filter and detection stages, to estimate the effort in error detection for guaranteeing an adequate level of reliability and also provide hooks for fault injection to test the respective detection components. Thus we try to identify and estimate the impact of sensor failures early in the design process and also to provide the information needed to estimate the confidence in the sensor data during operation.

To achieve the goals, we make intensive use of specification and simulation techniques. Our primary tool is Matlab/Simulink to describe the behavior of sensors and the computational components that filter and evaluate the sensor data. This paper presents our framework for model-driven development of safety-related components and how it is integrated in Simulink.

The paper is organized as the following. First present the main objectives derived from the introduction. The framework for model-driven development is presented in the subsequent section. Then, we briefly discuss the state of the art. Finally, we conclude the presented framework and give an outlook of our work.

2 Objectives

The components involved in critical perception tasks are typically sensors, detectors and filters. Within the model-driven development, these devices are encapsulated as component. We consider failures in these components to increase the overall safety of control. For the model-driven development it is indispensible to provide an appropriate model and description of typical failures. This knowledge can be exploited in many ways in the the model-driven development. It is used to configure the failure injection procedures, analyze the failure propagation checking and allows to check whether application requirements match the provided level data validity. These advantages can only be achieved due to the additional knowledge about the component interaction. The interaction lists each link between components combined with constrains. The interaction in combination with the description specifies the anticipated set of failures that an application has to cope with. By using model-driven development either the processing chain of the perception can be shaped to meet application requirements or the application can be refined to cope with the perception quality. Failure injection techniques and mixed-reality systems support this adjustment. The following objectives summarize the mentioned steps.

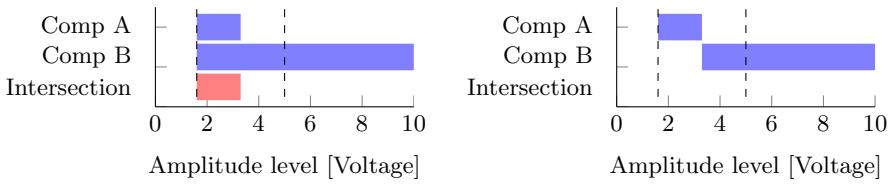
1. Specification and description of component characteristics
2. Definition and verification of component interaction
3. Mixed-reality and multi-target support
4. Seamless integration of failure injection techniques

3 Concept

The concept is structured into three parts according to the defined objectives: description, combination, and execution. First, we present a description form encompassing the component characteristics. The subsequent part deals with the combining and linking components based on model-driven development techniques. The advantages resulting from this part are the verification of the linked components and the specification of an anticipated set of failures. In contrast to usual solutions, these features are provided without on-line knowledge . So far, the online-knowledge is obtained by exhaustive simulations runs. The last development step, the execution, combines the description and the proven linkage of components in order to support mixed-reality and multi-target systems. Further, a model generator embedded in the execution phase automatically integrates failure injection techniques. The main objective of failure injection is to estimate the effect of failures in realistic simulation runs rather than just analyzing these effects [4, 5].

3.1 Description

The description informs about interfaces and characteristics of a component. This includes the input and the output interfaces of a component. A unique set of interface parameters enrich the interface description. Interface parameter specify the operational condition under which a component will work for certain. In a later development phase named design phase, those interface parameters will be configured and checked. To achieve this checking, the set of interface parameters have to be structured identically. In fact this is due to mathematical foundations, where operands can only be compared if they belong to the same type. In Fig. 1, the intersection of exemplary interface parameters is shown. In



(a) partly intersecting ranges (b) no intersection

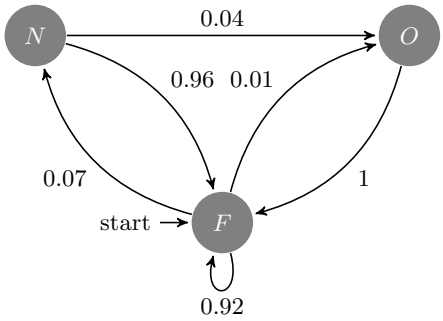
Fig. 1: Interface parameter checking

this example, the linked interfaces of component A and component B are checked against each other. In the first example the interface parameters are compatible due to intersection, which is highlighted by the red beam depicted in Fig. 1a. The second example presents incompatible interface parameters as shown in Fig. 1b. We underline that the presented approach for parameter checking is not limited to voltage ranges.

Regarding characteristics, the description consist of a functional part and a non-functional part. Access to the behavior of a component is provided by the functional part. All functionalities in Simulink are represented by blocks. Such a block might perform a simple unary operation like a summation or a complex algorithm to analyze wavelets. So, the complexity of a single block varies heavily and supports interfaces as well as function calls to cover the behavior of a component. Because of this importance, we refer to Simulink blocks within the function description. In detail, a referred Simulink block could be part of the Simulink library, a customized s-function or a Simulink model. Within the functional description, at least one Simulink block needs to be referred. Otherwise, this component would have no behavior. Considering the frequently used execution context in model-driven development processes, the simulation, Software in the Loop (SIL), Hardware in the Loop (HIL) or final target request for a respective functional description. The set of functional descriptions will be used to automatically switch the execution context depended on development demand.

The non-functional part of the description instructs about the deviation of a component from the nominal operation behavior. Such a deviation might lead to a non-negligible impact on the behavior of following components. As a result, the assured safety integrity level of the entire system might be threatened. In order to identify such discrepancies, the non-functional characteristics of each component are described. The potential benefit for the model-driven development are the configuration of failure injection procedures, the failure propagation checking and the requirement fulfillment checking. First, the configured failure injection leads to more precise simulation and SIL results due to the consideration of deviations. Second, the failure propagation checking assesses the usefulness of failure handling in advance. For instance, a sensor shows outlier failures and the following failure handling is not dealing with this particular failure, it can be assumed that the resulting information set will also contain outlier failures. Third, the knowledge about the non-functional characteristics could be exploited to estimate the possibility to fulfill requirements.

In detail, probable failures are described by the non-functional characteristics. This set of failures could be elaborated by either an analytical breakdown or an empirical research. Each elaborated failure type is listed in the description in combination with an occurrence probability and a deviation distribution function. The likelihood of a failure is described by the occurrence probability. The deviation distribution function contains the severity of a failure. In fact, to provide suitable non-functional characteristics the elaboration must be carefully determined. Fig. 2 shows a statistical model of the occurrence probability of an exemplary set of failures. This example respects the failure-free (F), outlier fail-

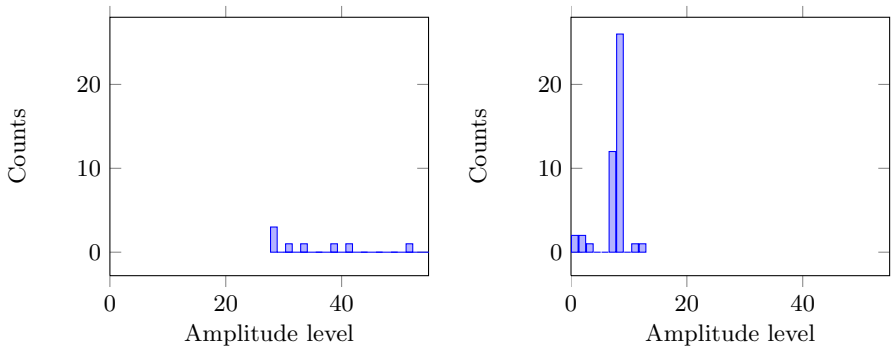


(a) Markov model representing failure types

$$\begin{bmatrix} 0.92 & 0.07 & 0.01 \\ 0.96 & 0 & 0.04 \\ 1 & 0 & 0 \end{bmatrix} = prob$$

(b) Corresponding transfer matrix (prob)

Fig. 2: Statically modelled non-functional characteristics considering failure types (failure free (F), outlier (O), noise(N))



(a) Characteristic of the error amplitude for outliers

(b) Characteristic of the error amplitude for noise

Fig. 3: Error level of two examplary error types

ure (O) and noise (N) state. In Fig. 2a,we visualize a markov chain representing the statistical behavior of this set of failures. The corresponding non-functional description is given in Fig. 2b as a transfer matrix (prob). In order to illustrate the severity of the mentioned set of failures, we plot the deviation distribution function in Fig. 3.

A general description of non-functional characteristics regarding sensor components could not be made. Because the set of elaborated failures depend heavily from the applied sensor. Several sensors show more frequent sporadic failures like outlier like depicted in Fig 3a. On the other side sensors suffer under noise failures as shown in Fig 3b. Therefore the concrete set of failures needs to be individually evaluated [6]. Considering detection components the set of failures consist only of false positives and false negatives [7]. In this case false positives

describe erroneously detected failures by the detection component. In contrast, false negatives describe not detected failures even though failures are present. Filter components can best be described [8] by failure impact parameters, which define the failure transfer to be static, proportional or lead to an elimination of a failure.

3.2 Design

The design phase is prepended to the general development phase and models the interaction between components. First of all, components are selected and linked together via interfaces. Then, the interface parameters of the linked interfaces are configured. To guarantee a compatible design, the interfaces and the interface parameters are checked. By using a regular grammar, the composition of components are checked. This is facilitated by uniquely defined components. In detail, we distinguish between sensor (S), detection (V), filter (V), application (P) and communication (C) components. Smart sensor designs are verified by the regular expression shown in equation 1. This expression could generate combinations of sensors (S) connected with detection or filter components (V), followed by exactly one application (P) and ending by a communication component (C). Moreover, the equation 2 specifies the regular expression to verify smart transducer designs, where communication components are accepted to be an input.

$$(S|(SV)) + P(C)+ \tag{1}$$

$$(S|C|(SV)|(CV)) + P(C)+ \tag{2}$$

The checking of interface parameters is based on mathematical expressions. Interface parameters of two components must intersect each other to be compatible. An interaction between components is stated to be compatible if the design could be built with the regular grammar and the configured interface parameters match each other. Compatible interfaces lead to a valid working set of components, which are recorded in a specification description inspired by EAST-ADL2 [9].

The second purpose of the design phase is to check non-functional characteristics. Because the previously described interface checking guarantees only the ability of components to work together, but nothing have been done to account on failures. At this point, the non-functional characteristics gets important and informs about failures. By exploiting that knowledge, the failure handling is traced and the fulfillment of requirements is checked. To trace the failure handling, the propagation of failures through the processing chain is determined. The processing chain is given by the defined interaction of the components, as described above. In order to determine the propagation, the non-functional characteristics are mapped to mathematical terms. These mathematical terms are formed to an expression by the usage of tailored operators defined in [8]. The resulting value of this expression reflect the anticipated set of failures an

application has to cope with. In addition to the trace of failure handling, the result is also used to check the fulfillment of requirements in advance. All of the mentioned features are achieved without execution of the model, which support the compatibility of the design and the fulfillment of requirement in very early development stage.

After the design phase satisfies the compatibility and the requirements, the execution phase is ready to perform.

3.3 Execution

In general, the execution phase corresponds to the originally way of model-driven development, where the behavior of an application is modeled and later on transformed to an executable program. This transformation process is commonly done by a code generator. But due to development, design, debug, test or verification reasons, the target is variable instead of static wherefore the code generation was initially designed. In this case, the modeled application needs continuously adjusted in order to support simulation, SIL, HIL or a final targets. By only using the code generation approach, the interfaces, compiler sets and device drivers needs to be manually switched. Those switches are error-prone and might threaten the correctness of the generated code. In the best case, the compiler reveals an incompatibility. Otherwise the engineer is responsible to avoid safety-critical situations.

The described situation could be avoided by a more general approach. Instead of adjusting interfaces, data structures and compiler sets manually, we propose a model generator take care of these error-prone issues. The merit of this automatism lies in the description and in the design phase. Without the functional characteristics given by description, an automatized target switch would be impossible. Because functional characteristics inform about the required Simulink block to employ the respective component within the desired execution context (simulation, SIL, HIL). Moreover, the design description includes a proven set of interface parameters needed to configure the Simulink block.

The generation and configuration of functional characteristics is only part of the solution. Regarding simulations, SIL and HIL, failure injection techniques need to be considered in order to keep the model execution as realistic as possible. Without failure injection techniques, an application is feed by a set of generated sensor information, which correspond to an ideal observation and those do not belong to a set of sensor information from real sensors. This loss of realism leads to simulation results showing the application behavior under ideal conditions. In this case, the application might show the intended behavior within simulation runs. But those results are not valid for sensor information, which are obtained from a real sensor and are superimposed by failures.

To employ failure injection procedures, the knowledge about the set of failures is needed. Moreover how often those failures should appear and which impact a respective failure should have. All of these information are hold by the non-functional description of each component. For failure injection purpose, the set of failures is taken together with their respective occurrence probabilities. To

determine which failure should be injected, the failure probabilities are mapped to ranges of a distribution function. These ranges have to correlate to the occurrence probabilities of the set of failures. Then a random number is taken, a particular failure will be injected if the random number falls into a range corresponding to that failure. This approach could be compared to a bag filled with items. The items inside the bag are marked that they belong to a certain failure type. The quote of marked items correspond to the occurrence probability. After the failure injection is triggered, the same approach is invoked to assign the severity of the failure.

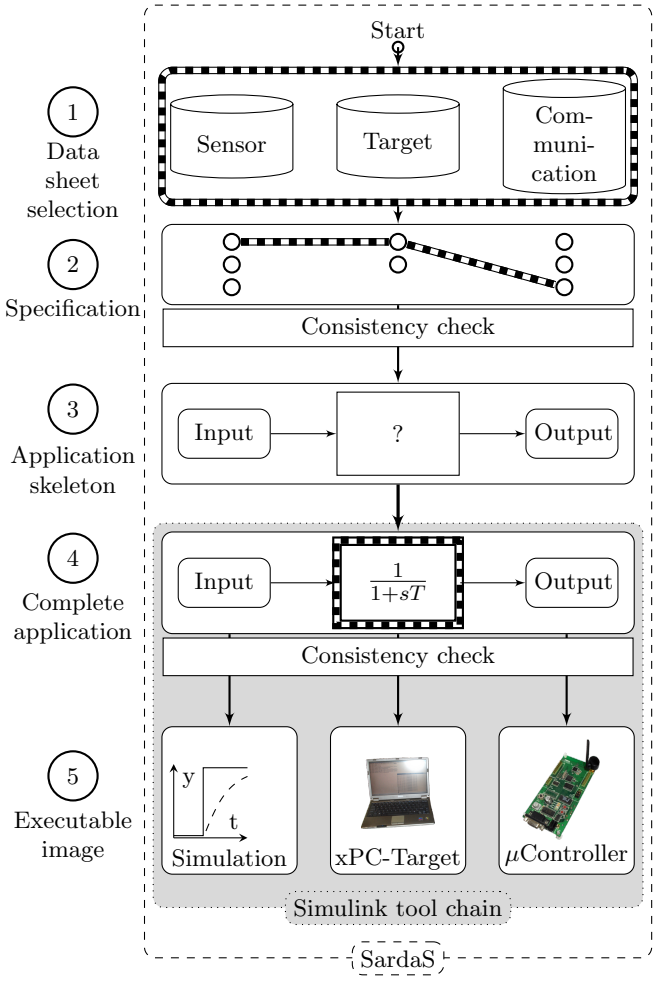


Fig. 4: Structure of our Simulink based framework

3.4 Summary

The explained model-driven development process is summarized in Fig. 4. First, for each component the description needs to be characterized. For the sake of clarity, we only consider sensor, target and communication components. The respective description is used to generate interfaces, which will be linked by an engineer. Afterwards, the Simulink framework checks interfaces for compatibility reasons and performs failure propagation checking. In the third step, an application skeleton including configured device drivers, compiler settings and data structures is generated. An application is linked against the generated functionalities depicted as step 4. Multiple execution contexts are supported due to the functional description. In this example, we consider simulation loops, HIL using an x-PC and microcontroller targets. Failure injection techniques are added to simulation and HIL context in order to feed the application with realistic input. Even the seamless integrated failure injection makes model-driven development feasible for critical perception.

4 State of the art

The state of the art is subdivided into four parts related to the defined objectives.

Specification and description of component characteristics - The characteristics of safety-related components can be determined either by an analytical breakdown or an empirical evaluation. Regarding the analytical approach, sensors could be analyzed this way as shown in [10]. Given a transfer function a filter component is clearly characterized as indicated in [11]. Salicone [12] proposed a way to trace the uncertainty of an application using random-fuzzy variables. Concerning empirical research, uncertainty values specify sensor components as shown in [13]. Confidence intervals are used to define bounds on the sensor data [14]. An empirical approach exists characterizing detection components along the terms false positive, false negative, true positive and true negative [15].

Sensor ML [16] is an approach that may be used to describe the considered components. This techniques sacrifices the interoperability showing large flexibility to describe interactions. But particularly interoperability is a need for model-driven development. IEEE 1451 [17] satisfies the interoperability needs but is limited to sensor components. OMG STI [18] also provides interoperability but it is only applicable to communication components. A description technique encompassing all components by providing the interoperability cannot be found.

Definition and verification of interacting components - A technique to define the interaction between components is known as EAST-ADL2 [9]. Autosar [19] also provides the relations and interactions between components in combination with a late binding of the used interfaces. After the linkage between components has been established, potential failures can be traced by FEMA [5] and FTA [4] techniques.

Seamless integration of failure injection techniques - Failures can be injected either in hardware, software or in models-based designs. Pin-probes [20] or customized sockets establish a physical connection to inject failures in hardware. A common contactless techniques uses radiation and electromagnetic inferences [21]. VHDL could be exploited to reconfigure logic cells in FPGA designs [22]. Regarding software-based failure injection, the memory map could be switched. Further, exception-based mechanisms and debugging interfaces [23] are also known as software-based failure injection. A comprehensive model-based failure injection framework is introduced in [24]. The referred failure injection techniques come with respective pros and cons regarding failure injection properties like reachability, controllability, repeatability, etc. For further classifications, the interested reader is directed to [24].

Mixed-reality and multi-target support - The generation of 3D-robotic environments out of textual descriptions is given by [25]. Model-driven architecture [26] provide transformation techniques for multi-target support. A code generation support for model-driven development processes is known as target language compiler (TLC) embedded in [27]. The translation of graphical programming techniques to executable code is presented in [28]. A standardized architecture called Autosar [19] achieves multi-target support by a component-based design model.

Table 1: Covered objectives by the state of the art

		SensorML	IEEE 1451	OMG STIS	Simulink	LabView	Autosar	Modifi	FMEA/FTA	EAST-ADL2
1	Specification and description of component characteristics	X	X	X					X	X
2	Definition and verification of interacting components	X					X			X
3	Seamless integration of failure injection techniques							X		
4	Mixed-reality and multi-target support				X	X	X			X

5 Conclusion

We proposed a framework to develop systems that deal with perception in safety critical applications. We introduce a new description technique to deal with the

specific properties of the computational components involved in the perception task. The well-defined and described knowledge about the components serves as a base for model-driven development techniques. Further, these techniques are enhanced by a model generator to support failure injection. Our technique is suited for supporting mixed-reality systems as well as multi-target scenarios. Regular expressions verify the presented automatisms. The early evaluation of anticipated failures without model execution is a feature. That so far, is not available in similar frameworks. None of them is able to manage all objectives. Of course, the considered objectives could be mastered manually. But this error-prone adjustments are not verified and might therefore threaten the safety of the system.

Acknowledgment

This work has been supported by the EU under the FP7-ICT programme, through project 288195 "Kernel-based ARchitecture for safetY-critical cONtrol" (KARYON).

References

1. C. Geyer, S. Singh, and L. Chamberlain, "Avoiding collisions between aircraft: State of the art and requirements for uavs operating in civilian airspace," *Robotics Institute, Carnegie Mellon University, Tech. Rep. CMU-RI-TR-08-03*, 2008.
2. K. Marzullo, "Tolerating failures of continuous-valued sensors," *ACM Transactions on Computer Systems (TOCS)*, vol. 8, no. 4, pp. 284–304, 1990.
3. K. Ni, N. Ramanathan, M. N. H. Chehade, L. Balzano, S. Nair, S. Zahedi, E. Kohler, G. Pottie, M. Hansen, and M. Srivastava, "Sensor network data fault types," *ACM Transactions on Sensor Networks (TOSN)*, vol. 5, no. 3, p. 25, 2009.
4. W. E. Vesely and N. Roberts, *Fault tree handbook*. Nuclear Regulatory Commission, 1987.
5. D. H. Stamatis, *Failure Mode and Effect Analysis: FMEA from Theory to Execution*, 2nd ed. ASQ Quality Press, 4 2003.
6. S. Zug, A. Dietrich, and J. Kaiser, "An architecture for a dependable distributed sensor system," *IEEE Transactions on Instrumentation and Measurement*, vol. 60 Issue 2, pp. 408 – 419, 2 2011.
7. T. Brade, J. Kaiser, and S. Zug, "Expressing validity estimates in smart sensor applications," *ARCS 2013*, 2013.
8. S. Zug, T. Brade, J. Kaiser, and S. Potluri, "An approach supporting fault-propagation analysis for smart sensor systems," in *Computer Safety, Reliability, and Security*. Springer, 2012, pp. 162–173.
9. A. Sandberg, D. Chen, H. Lönn, R. Johansson, L. Feng, M. Törngren, S. Torchiaro, R. Tavakoli-Kolagari, and A. Abele, "Model-based safety engineering of interdependent functions in automotive vehicles using east-adl2," in *Computer Safety, Reliability, and Security*. Springer, 2010, pp. 332–346.
10. Y. Huang, J. Gertler, and T. J. McAvoy, "Sensor and actuator fault isolation by structured partial pca with nonlinear extensions," *Journal of Process Control*, vol. 10, no. 5, pp. 459–469, 2000.

11. R. Isermann, *Digital control systems: vol. 2: stochastic control, multivariable control, adaptive control, applications*. Springer-Verlag New York, Inc., 1991.
12. S. Salicone, *Measurement Uncertainty: An Approach Via the Mathematical Theory of Evidence*, ser. Springer Series in Reliability Engineering. Springer Science+Business Media, LLC, 2007.
13. R. Moffat, "Contributions to the theory of single-sample uncertainty analysis," *ASME, Transactions, Journal of Fluids Engineering*, vol. 104, pp. 250–258, 1982.
14. H. Kopetz, M. Holzmann, and W. Elmenreich, "A universal smart transducer interface: Ttp/a," in *Object-Oriented Real-Time Distributed Computing, 2000.(ISORC 2000) Proceedings. Third IEEE International Symposium on*. IEEE, 2000, pp. 16–23.
15. J. Davis and M. Goadrich, "The relationship between precision-recall and roc curves," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 233–240.
16. Open Geospatial Consorium Inc., *OpenGIS Sensor Model Language (SensorML) Implementation Specification Version 1.0.0*, 2007.
17. IEEE Standards Association, *IEEE Standard for a Smart Transducer Interface for Sensors and Actuators (IEEE 1451.2)*, 1997. [Online]. Available: http://ieeexplore.ieee.org/xpl/standards.jsp?findtitle=1451&letter=1451
18. Object Managment Group (OMG), *Smart Transducer Interface Specification*, 7 2003.
19. Autosar Consortium, "Standard Specifications," 2013. [Online]. Available: \url{http://www.autosar.org/index.php?p=3&up=1&uup=1&uuup=0}
20. J. Arlat, Y. Crouzet, and J.-C. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems," in *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*. IEEE, 1989, pp. 348–355.
21. F. Vargas, D. Cavalcante, E. Gatti, D. Prestes, and D. Lupi, "On the proposition of an emi-based fault injection approach," in *On-Line Testing Symposium, 2005. IOLTS 2005. 11th IEEE International*. IEEE, 2005, pp. 207–208.
22. E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into vhdl models: the mefisto tool," in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*. IEEE, 1994, pp. 66–75.
23. M. Portela-Garcia, C. López-Ongil, M. García-Valderas, and L. Entrena, "A rapid fault injection approach for measuring seu sensitivity in complex processors," in *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*. IEEE, 2007, pp. 101–106.
24. R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren, "Modifi: a model-implemented fault injection tool," in *Computer Safety, Reliability, and Security*. Springer, 2010, pp. 210–222.
25. R. Diankov, T. Adviser-Kanade, and J. Adviser-Kuffner, *Automated construction of robotic manipulation programs*. Carnegie Mellon University, 2010.
26. R. Soley *et al.*, "Model driven architecture," *OMG white paper*, vol. 308, p. 308, 2000.
27. The Mathworks, "Simulink - Webseite," 2011. [Online]. Available: \url{http://www.mathworks.de/products/simulink/}
28. National Instruments, "LabVIEW 2009 - Herstellerwebseite," 2009, verfügbar unter http://www.ni.com/labview/d/ am 20.02.2011.