Towards Digital Twins for the Description of Automotive Software Systems

Jan Olaf Blech
Aalto University, Finland
jan.blech@aalto.fi

We present models for automotive software that capture quantitative and qualitative aspects of software systems and the underlying hardware architecture. In particular, we consider different levels of computing power. These range from controllers up to the cloud. We present a modeling approach for software deployment taking different automotive requirements such as criticality, latency, memory, computational resources, and communication into account. Our models capture automotive software and hardware system configurations and can serve as digital twins that are digital counterparts of (usually) physical entities. Furthermore, we highlight connected research areas and challenges.

1 Introduction

In the past decades, software in the automotive domain has gained an increasingly important role: functionality that used to be realized by electronic, electrical and mechanical devices alone is now frequently controlled by software. Software can run on more than 70 ECUs (Electronic Control Units)[5] inside a car. In recent times, a consolidation of software and ECUs, i.e., the replacement of several microcontrollers by using a more powerful ECU, has gained increased attention. In addition new software-based functionality and the introduction of additional computational resources such as cloud-computing, have found their way into the automotive world. This trend is complemented by the increased use of technology that has its origins in the IT-world such as Ethernet, general purpose computing devices and operating systems such as Linux.

This paper primarily motivates research challenges on models serving as digital twins with a special emphasis on quantitative aspects used in the partitioning of software across different ECUs and embedded computers in a car as well as cloud-based services. The term digital twin describes a digital counterpart of a (usually) physical entity such as a product (e.g., a car), a part of a product (e.g., a car's engine) or a machine (see Figure 1). Note, that digital twins can describe non-physical entities as well such as the software architecture of a system.

Here, we are using a notion of models that goes beyond traditional models in model-based development. Models are abstractions of systems and can include aspects such as physical characteristics, geometrical layout, wiring, communication links, hardware resources (e.g., computational power) and information on the structure of software such as components, layers and interfaces. Models of systems such as cars can be used at design and development time, during the car's life time and even during and after the decommissioning phase. They can be used for design, documentation, optimization purposes and present an abstraction of the system and thus serve as digital twins of automotive hardware and software.

In this paper, we use the following ingredients for interconnected computational devices: *Microcontroller*, as devices that are primarily developed to interact with sensors and actuators. *Embedded*

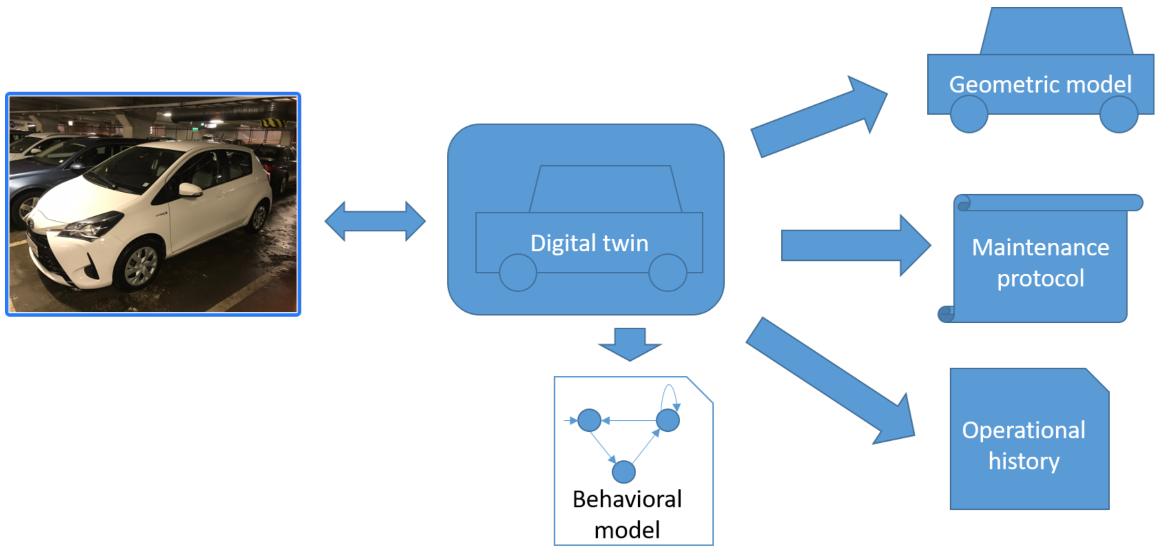

Figure 1: Digital Twin

computers, as devices that do typically feature classical IT components such as general purpose microprocessors, RAM/microprocessor on separate chips and ethernet. The user of a *cloud* service is in general unaware of the actual location and resources where the service is provided. In principle services can be shifted freely between data-centres on different continents. This full flexibility, however, may cause unpredictable delay times. The strategy to move services closer to the actual controller is sometimes referred to as fog computing [4]. If the services are located at the outer edge of the network the term edge-computing is used [17].

2 Related work and Research Areas

Typically software running on an ECU is structured in layers such as base software including drivers, and a runtime environment (RTE) which enables the deployment of application software. Currently, the AUTOSAR Classic standard[1] and Posix-based operating systems are mainly used. In the rather monolithic AUTOSAR Classic standard, the application software itself is structured into components which may further be structured into subcomponents that comprise executable code units (called runnables in AUTOSAR). Figure 2 shows the AUTOSAR Classic stack. It can be seen that all communication between different applications and services from the base software such as network communication is performed via the RTE. All communication pathes through the RTE are statically generated and can not adapt after compile time.

Posix-like operating systems such as the adaptive AUTOSAR framework (see, e.g., [11]) allow a greater amount of reconfiguration, typically require more resources and may have trouble to meet certain other criteria such as reliability and latency. Figure 3 presents the architectural view of an adaptive AUTOSAR systems which is realized on-top of a Posix operating system such as Linux. In Posix systems processes (realizing applications) can be started, stopped, loaded and terminated at run-time of a system.

Mixed Criticality systems is a research area connected to our work (see e.g., [7]) and partitioning of

[1]`https://www.autosar.org/standards/classic-platform/`, retrieved 30th November 2018

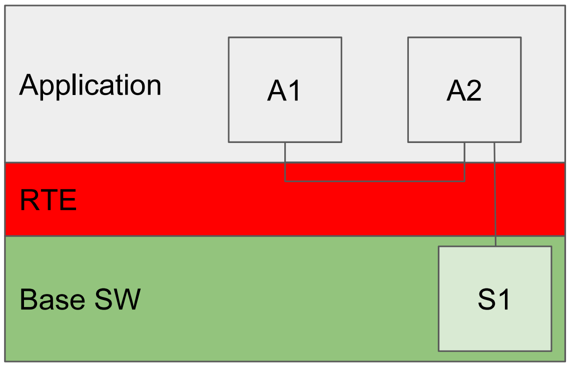

Figure 2: AUTOSAR Classic Software Stack

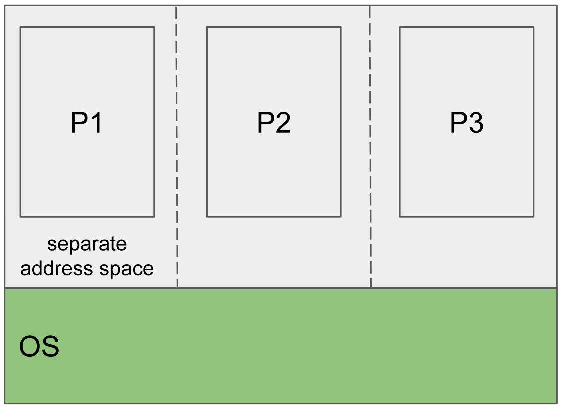

Figure 3: Adaptive AUTOSAR (Posix) Software Stack

software has been addressed in [10]. Scheduling is frequently regarded as an important aspect of Mixed Criticality Systems (see, e.g. including multi-core challenges [12]). Task scheduling in this context has been, e.g., studied in [13]. Design Space Exploration in the ISO 26262 context has been studied in [16] and regards the partitioning of software components to hardware elements. The paper focuses on criticality aspects of software functions (i.e., ASIL level classification).

Hardware-Software Co-Design (e.g., [9] and [8]) regards the question whether a functionality should be realized in hardware of software, or if there is a possibility to further partition it into hardware and software parts. Design-space exploration techniques have been applied to this domain as well.

The need for clarification of terminology in the automotive software world was proposed in [6] together with a service hierarchy proposition for embedded automotive software. Behavioral types [3, 18] are a related formalism that we have introduced to describe state-based properties of both software and cyber-physical systems. The work described in this paper continues this view by presenting models for hardware and software systems. State-based behavior can be annotated using the properties featured in our models. We have investigated the use of models serving as digital twins together with a cloud-based service infrastructure in the industrial automation domain [2]. An emphasis here was on the remote response to incidents in industrial facilities.

3 Models Mirroring the Car

Digital twins as digital mirrors of (mostly physical) entities have gained popularity in the manufacturing domain [15] especially in the Industrie 4.0 [14, 1] context. Models can serve as digital twins throughout the development, construction and commissioning, life-time, and decommissioning of a technical system. Meta-models are the datatypes for models. They outline what information can be kept inside a particular model. Models can be stored in the cloud and a variety of mechanisms have been developed to maintain and use models in various stages and tools, such as the open source Eclipse Modeling Framework [2] that comes with modeling, tool-support, parsers and cloud-based storage mechanisms.

While we put an emphasis on software partitioning at development time, the models for digital twins introduced in this paper (both software and hardware models) can also serve the following purposes:

- Reconfiguration (see e.g., [19] for work on model support in this context) of software at runtime (during the operation of a car). This means, that we answer the question which application software component should run on which ECU.
- Diagnosis, detecting malfunctions and identifying which software components are affected.
- Resolving issues at run-time. For example, if a software component cannot run on a particular ECU anymore (e.g., due to a hardware failure), its functionality could be shifted to another ECU or another software component could provide backup functionality.
- Verification and validation of constraints such as requirements on software architecture.
- Tracking of changes that occur during the live-time of a system. Systems may evolve during the lifetime. Maintenance protocols including the replacement of ECUs and the update history of software can be archived using models.
- Tracking and helping to gain relevant information during the decommissioning of a system: the history of a system may provide useful hints on how to best decommission a system.

Figure 4 shows the organization of ECUs in a car using three hierarchical levels plus the cloud. Shifting functionality between these levels is one goal that we want to draw attention to.

4 Partitioning of Software

In this paper, we are particularly interested in the question where to deploy a piece of software that realizes a functionality. In order to find an optimal location, we need to develop models that capture both the system including ECUs and physical buses as well as logical connections between different software components. Here, we present a mathematical founded model for both hardware and software.

4.1 Proposed Hardware Model

All devices on which software can run (*ECU*s) and their interconnections are modeled as a graph $(ECU, NL, \mathscr{P}_{\mathscr{HW}})$ comprising a set of hardware devices *ECU*, a set of communication links $NL : ECU \times 2^{A_{HE}} \times ECU$ and a function $\mathscr{P}_{\mathscr{HW}} : ECU \mapsto 2^{A_{ECU}}$ ($2^{A_{ECU}}$ denotes the powerset of A_{ECU}). Note, that the set *ECU* is quite general in order to capture all kinds of computational devices: Cloud-based services are in *ECU* as well as controllers, embedded computers and smart sensors.

[2]`https://www.eclipse.org/modeling/emf/`

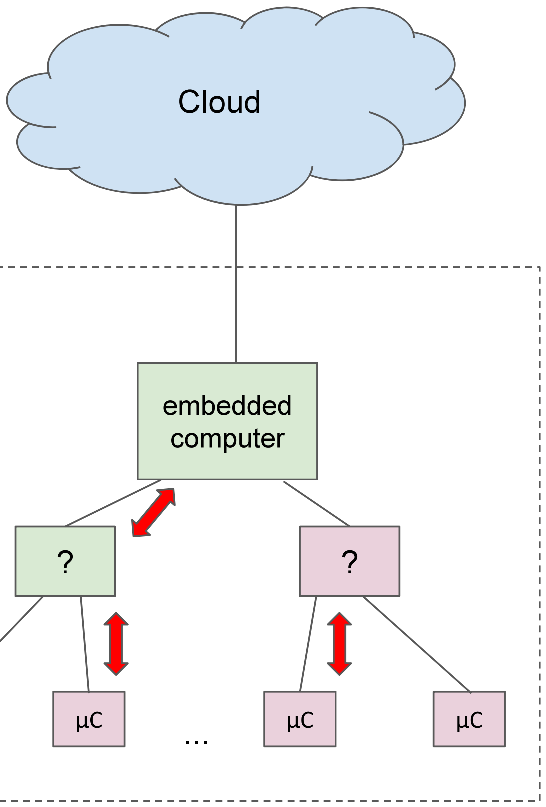

Figure 4: Re-Partitioning of Software

Communication links formalize hardware-based links such as buses, but also wireless communication channels between members of the *ECU* set. A communication link $(ecu_i, a, ecu_j) \in NL$ comprises a source communication device ecu_i a set of attributes $a \in 2^{A_{HE}}$ that comprises attributes such as speed and capacity and a target device ecu_j. Figure 5 shows a simple example for an automotive bus structure. ECUs are connected via Ethernet, Flexray, CAN and LIN buses. The figure illustrates that there can be more than one connection between two ECUs. Note, that the actual communication messages that can be sent between different ECUs are typically fixed as at an early development stage. Therefore, during the development we effectively have a one-to-one communication between different ECUs which is captured in our graph formalization.

For reasoning about a system, we need to describe properties of both computational devices and communication links. These properties are described as attributes. The link attributes are directly contained in the link, for *ECUs* the function $\mathscr{P}_{\mathscr{HW}}$ maps hardware devices to a set of attributes from A_{ECU}. Typical attributes from A_{ECU} comprise:

- Computational power, such as the number of available cores, their speed, their architecture.
- Hardware-Architectural features such as lockstep computation, special co-processors.
- Memory characteristics such as RAM/ROM/Flash, on-chip, separate chips, size and speed.
- IO capabilities such as hardware interfaces, available sensors and actuators, general purpose IO pins.

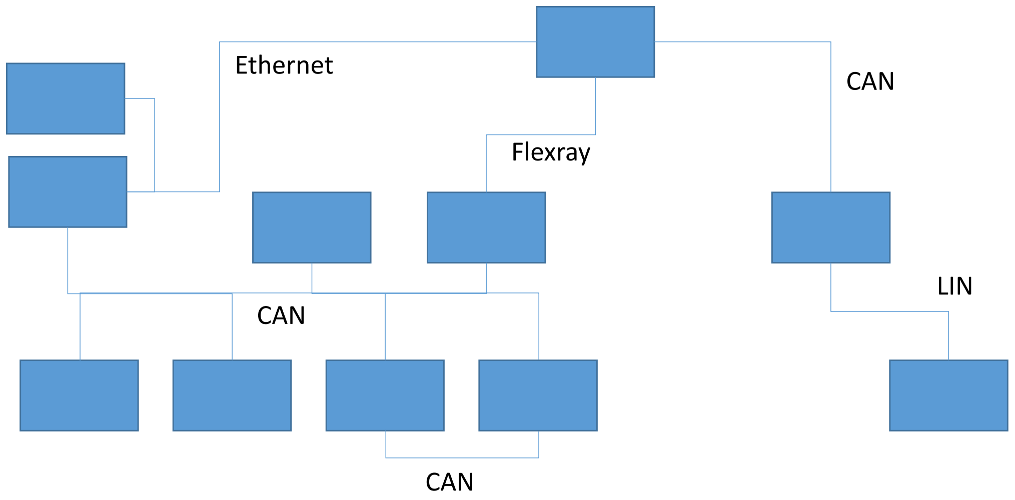

Figure 5: A Simplified View on Automotive Buses

- Networking capabilities, such as interfaces to specific bus and other communication infrastructure

4.2 Proposed Software Model

The key idea for describing a software system is to describe the application components and necessary communication between them. The software system is described as a graph $(SWC, E, \mathscr{P}_{\mathscr{SW}})$ featuring a set of Nodes SWC, a set of edges $E : SWC \times 2^{A_{SE}} \times SWC$ and a function $\mathscr{P}_{\mathscr{SW}} : SWC \mapsto 2^{A_{SWC}}$ that maps nodes to attributes. Our model features the following abstractions:

- Software is broken down into atomic components and each atomic software component becomes a node $n_i \in SWC$.
- Nodes are associated with attributes to indicate resource and other hardware requirements such as RAM consumption, communication requirements resulting in links and upper bounds on communication time to actuators and sensors as well as the criticality of the functionality realized by the software component. This is realized using the function $\mathscr{P}_{\mathscr{SW}}$ which maps a node to a set of relevant attributes $2^{A_{SWC}}$
- Edges $(n_j, a, n_k) \in E$ represent the required communication between atomic software components. Each edge is associated with a set of attributes $a \in 2^{A_{SE}}$. Typical attributes are the amount and nature of communication and latency requirements.

Note, that we only look at application components in our model and assume that typical requirements on the base software and run-time environments can be realized and thus exclude them from our model. In order to achieve full digital twins, additional attributes can be added to track, e.g., physical locations and maintenance protocols.

4.3 An Example System

To give an idea of how our models can be used, we provide a small example system. The hardware model is given: by a set of computational devices: $ECU = \{Cloud, GW, C1, C2\}$, comprising the cloud $Cloud$, a gateway device realized as an embedded PC GW and two controllers $C1$, $C2$.

We only regard three properties in this example: RAM, the local availability of a sensor *Sen* and an actuator *Act*. Thus, the property function $\mathscr{P}_{\mathscr{HW}}$ is defined as:

$\mathscr{P}_{\mathscr{HW}}(Cloud) = \{''RAM == infinity''\}$
$\mathscr{P}_{\mathscr{HW}}(GW) = \{''RAM == 1024MB''\}$
$\mathscr{P}_{\mathscr{HW}}(C1) = \{''RAM == 4MB'', Sen\}$
$\mathscr{P}_{\mathscr{HW}}(C2) = \{''RAM == 2MB'', Act\}$

This means that the cloud has inifinity RAM ressources, the gateway has 1024 MB, the controller $C1$ has a local sensor *Sen* available and 4 MB RAM and the controller $C2$ has an actuator *Act* and 2 MB RAM. The set of communication links is defined as:
$\{(Cloud, \{\}, GW), (GW, \{\}, C1), (GW, \{\}, C2),$
$(GW, \{\}, Cloud), (C1, \{\}, GW), (C2, \{\}, GW)\}.$
Links between the cloud and the gateway as well as between the gateway and the controllers are contained. No properties are given in this simple example. Note, that we need to include both directions for bidiractional communication.
The software model is defined as follows:

1. The set of nodes, representing software components:
   $SWC = \{CtrlS, CtrlA, Comp1, Comp2, Comp3\}$.
   We have five software components: *CtrlS* is the control component for a sensor, *CtrlA* a control component for an actuator, *Comp*1, *Comp*2 and *Comp*3 are other components performing computations or offering services.

2. Required communication is specified as the set:
   $\{(CtrlS, \{\}, Comp1)$, $(Comp1, \{\}, CtrlA)$,
   $(Comp1, \{\}, Comp2)$, $(Comp1, \{\}, Comp3)\}$.
   One can see that the *CtrlS* component needs to communicate with *Comp*1. *Comp*1 needs to communicate with *CtrlA*, *Comp*2 and *Comp*3.

3. The properties of the software components are defined by instantiating the property mapping function $\mathscr{P}_{\mathscr{SW}}$:
   $\mathscr{P}_{\mathscr{SW}}(CtrlS) = \{Sen\}$ $\quad \mathscr{P}_{\mathscr{SW}}(Comp2) = \{\}$
   $\mathscr{P}_{\mathscr{SW}}(CtrlA) = \{Act\}$ $\quad \mathscr{P}_{\mathscr{SW}}(Comp3) = \{\}$
   $\mathscr{P}_{\mathscr{SW}}(Comp1) = \{\}$
   The function is used to state requirements of the software components. Here, we have only formalized that *CtrlS* requries the local availability of the sensor *Sen* and *CtrlA* requires the local availability of the actuator *Act*.

4.4 Goals

The introduced models can be used to achieve at least two different goals when reasoning about good software partitoning:

1. Finding the right partitioning for a given software system and a given hardware system.
2. Finding a good hardware model for a given software system.

To solve these tasks, we define a solution quality evaluation function:
$\mathscr{E}$:
$(ECU, NL, \mathscr{P}_{\mathscr{HW}}) \times (SWC, E, \mathscr{P}_{\mathscr{SW}}) \times$

$(SWC \mapsto ECU) \mapsto int$
which takes a hardware model and a software model, furthermore it takes a function $SWC \mapsto ECU$ mapping an atomic software component to an ECU. The function returns a numerical value indicating the quality of the solution. Higher numbers indicate better solutions. Negative numbers indicate that no solution has been found.
In our example, the mapping $M : (SWC \mapsto ECU)$ given as:

$M(CtrlS) = C1$	$M(Comp2) = GW$
$M(CtrlA) = C2$	$M(Comp3) = GW$
$M(Comp1) = GW$	

may achieve a satisfying score.The $CtrlS$ is located on $C1$. Thus, the required proximity to the sensor is given. Likewise $CtrlA$ is located on $C2$ and the required proximity to the actuator is also provided. All other components are located on the gateway. With the formalized requirements, this is a feasible solution. Another alternative is given below:

$M(CtrlS) = C1$	$M(Comp2) = Cloud$
$M(CtrlA) = C2$	$M(Comp3) = Cloud$
$M(Comp1) = GW$	

Here, two of the $Comp$ components are located in the cloud. We would need to define more properties in order to determine if one solution is favorable over another.

Major research questions arise around good ways to find an appropriate mapping $M : (SWC \mapsto ECU)$. Typically one would use design-space exploration techniques (see above) to find a solution. A simple solution is to use search-based algorithms, and search through possible mappings, by simple trying them out. More intelligent techniques can use constraint-Solvers, e.g., SMT solvers, especially if one takes resources such as network capacity with numerical values into account. The task gets even more challenging if the hardware model is not fixed, but has to be discovered as well.

5 Conclusion

We presented models with an emphasis on the partitioning of software in and around cars as well as connected research challenges and classification of work. Our mathematical founded models are a way to formally represent digital twins and take different quantitative and qualitative aspects of systems into account. They can be used during all phases in the life-cycle of a car. However, in this paper, we particularly proposed the use during design and development time.

References

[1] Bauernhansl, T., Ten Hompel, M., & Vogel-Heuser, B. (Eds.). (2014). Industrie 4.0 in Produktion, Automatisierung und Logistik: Anwendung, Technologien und Migration (pp. 1-648). Wiesbaden: Springer Vieweg. doi:10.1007/978-3-658-04682-8

[2] Blech, J. O., Peake, I., Schmidt, H., Kande, M., Rahman, A., Ramaswamy, S., ... & Narayanan, V. (2015, September). Efficient incident handling in industrial automation through collaborative engineering. In Emerging Technologies & Factory Automation (ETFA), IEEE. doi:10.1109/ETFA.2015.7301533

[3] Blech, J. O., Falcone, Y., Rue, H., & Schätz, B. (2012, October). Behavioral specification based runtime monitors for OSGi services. In International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (pp. 405-419). Springer. doi:10.1007/978-3-642-34026-0_30

[4] Bonomi, F., Milito, R., Zhu, J., & Addepalli, S. (2012, August). Fog computing and its role in the internet of things. MCC workshop on Mobile cloud computing (pp. 13-16). ACM. doi:10.1145/2342509.2342513

[5] Broy, M. (2006, May). Challenges in automotive software engineering. In Proceedings of the 28th international conference on Software engineering (pp. 33-42). ACM. doi:10.1145/1134285.1134292

[6] Broy, M. (2010). Multifunctional software systems: Structured modeling and specification of functional requirements. Science of Computer Programming, 75(12), 1193-1214. doi:10.1016/j.scico.2010.06.007

[7] Burns, A., & Davis, R. (2013). Mixed criticality systems-a review. Department of Computer Science, Univ. of York, Tech. Rep, 1-69.

[8] De Michell, G., & Gupta, R. K. (1997). Hardware/software co-design. Proceedings of the IEEE, 85(3), 349-365. doi:10.1109/5.558708

[9] Eles, P., Peng, Z., Kuchcinski, K., & Doboli, A. (1997). System level hardware/software partitioning based on simulated annealing and tabu search. Design automation for embedded systems, 2(1), 5-32. doi:10.1023/A:1008857008151

[10] Ernst, R., & Di Natale, M. (2016). Mixed Criticality SystemsA History of Misconceptions?. IEEE Design & Test, 33(5), 65-74. doi:10.1109/MDAT.2016.2594790

[11] Fürst, S., & Bechter, M. (2016, June). AUTOSAR for connected and autonomous vehicles: The AUTOSAR adaptive platform. In Dependable Systems and Networks Workshop. IEEE. doi:10.1109/DSN-W.2016.24

[12] Herman, J. L., Kenna, C. J., Mollison, M. S., Anderson, J. H., & Johnson, D. M. (2012, April). RTOS support for multicore mixed-criticality systems. In Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th (pp. 197-208). IEEE. doi:10.1109/RTAS.2012.24

[13] Huang, J., Blech, J. O., Raabe, A., Buckl, C., & Knoll, A. (2011, October). Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (pp. 247-256). ACM. doi:10.1145/2039370.2039409

[14] Kagermann, H., Helbig, J., Hellinger, A., & Wahlster, W. (2013). Umsetzungsempfehlungen fr das Zukunftsprojekt Industrie 4.0: Deutschlands Zukunft als Produktionsstandort sichern; Abschlussbericht des Arbeitskreises Industrie 4.0. Forschungsunion.

[15] Rosen, R., Von Wichert, G., Lo, G., & Bettenhausen, K. D. (2015). About the importance of autonomy and digital twins for the future of manufacturing. IFAC-PapersOnLine, 48(3), 567-572. doi:10.1016/j.ifacol.2015.06.141

[16] Schätz, B., Voss, S., & Zverlov, S. (2015, June). Automating design-space exploration: optimal deployment of automotive SW-components in an ISO26262 context. In Proceedings of the 52nd Annual Design Automation Conference (p. 99). ACM. doi:10.1145/2744769.2747912

[17] Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. IEEE Internet of Things Journal, 3(5), 637-646. doi:10.1109/JIOT.2016.2579198

[18] Wenger, M., Zoitl, A., & Blech, J. O. (2015, September). Behavioral type-based monitoring for iec 61499. In Emerging Technologies & Factory Automation (ETFA). IEEE. doi:10.1109/ETFA.2015.7301447

[19] Yi, W. (2017, November). Towards Customizable CPS: Composability, Efficiency and Predictability. In International Conference on Formal Engineering Methods (pp. 3-15). Springer, Cham. doi:10.1007/978-3-319-68690-5_1