XANDAR: An X-by-Construction Framework for Safety, Security, and Real-Time Behavior of Embedded Software Systems

Tobias Dörr[1], Florian Schade[1], Juergen Becker[1], Georgios Keramidas[2,9], Nikos Petrellis[2], Vasilios Kelefouras[2], Michael Mavropoulos[2], Konstantinos Antonopoulos[2], Christos P. Antonopoulos[2], Nikolaos Voros[2], Alexander Ahlbrecht[3], Wanja Zaeske[3], Vincent Janson[3], Phillip Nöldeke[3], Umut Durak[3], Christos Panagiotou[4], Dimitris Karadimas[4], Nico Adler[5], Clemens Reichmann[5], Andreas Sailer[5], Raphael Weber[5], Thomas Wilhelm[5], Wolfgang Gabler[6], Katrin Weiden[6], Xavier Anzuela Recasens[6], Sakir Sezer[7], Fahad Siddiqui[7], Rafiullah Khan[7], Kieran McLaughlin[7], Sena Yengec Tasdemir[7], Balmukund Sonigara[7], Henry Hui[7], Esther Soriano Viguer[8], Aridane Alvarez Suarez[8], Vicente Nicolau Gallego[8], Manuel Muñoz Alcobendas[8], Miguel Masmano Tello[8]

[1]Karlsruhe Institute of Technology, Germany
[2]University of Peloponnese, Greece
[3]German Aerospace Center (DLR), Institute of Flight Systems, Germany
[4]AVN Innovative Technology Solutions Limited, Cyprus
[5]Vector Informatik GmbH, Germany
[6]Bayerische Motoren Werke Aktiengesellschaft, Germany
[7]Queen's University, Belfast, UK
[8]Fent Innovative Software Solutions, SL, Spain
[9]Aristotle University of Thessaloniki, Greece

***Abstract*—The safe and secure implementation of increasingly complex features is a major challenge in the development of autonomous and distributed embedded systems. Automated design-time procedures that guarantee the fulfillment of critical system properties are a promising approach to tackle this challenge. In the European project XANDAR, which took place from 2021 to 2023, eight partners developed an X-by-Construction (XbC) design framework to support developers in the creation of embedded software systems with certain safety, security, and real-time properties. The design framework combines a model-based toolchain with a hypervisor-based runtime architecture. It targets modern high-performance hardware, facilitates the integration of machine learning applications, and employs a library of trusted safety and security patterns to reduce the implementation and verification effort. This paper describes the concepts developed during the project, the prototypical implementation of the design framework, and its application in both an automotive and an avionics use case.**

***Index Terms*—X-by-Construction, model-based development, real-time systems, safety-critical systems, hypervisors**

I. Introduction

Modern embedded systems, such as self-driving cars or urban air taxis, are subject to a variety of requirements. Autonomous road vehicles, for instance, will often depend on nondeterministic Machine Learning (ML) algorithms for object detection and are integrated into a large-scale network. At the same time, they are subject to rigorous safety, security, and real-time requirements. This ongoing trend leads to challenging trade-offs in the development of embedded software systems. Self-driving cars, for instance, are required to exhibit fail-operational behavior [1], but they might employ powerful multicore processors, which can in turn increase the susceptibility to random hardware faults.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957210.

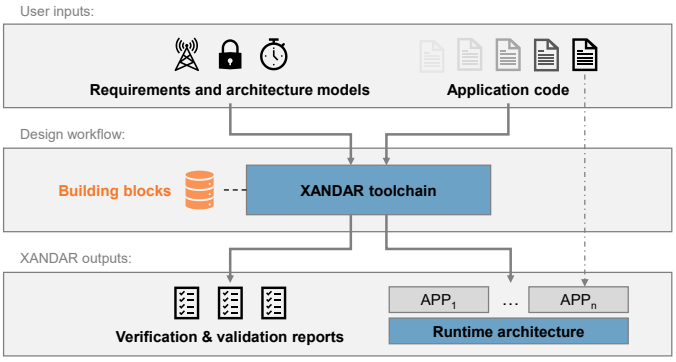

Fig. 1. Overview of the XANDAR design framework

In the European research project XANDAR [2], eight partners created a design framework that supports developers in fulfilling the safety, security, and real-time requirements of such systems. This paper presents the results that are available after the final stage of the recently completed project.

A high-level overview of the XANDAR design framework is visualized in Figure 1. The project partners designed, implemented, and evaluated two major results: a toolchain inspired by the X-by-Construction (XbC) paradigm and a

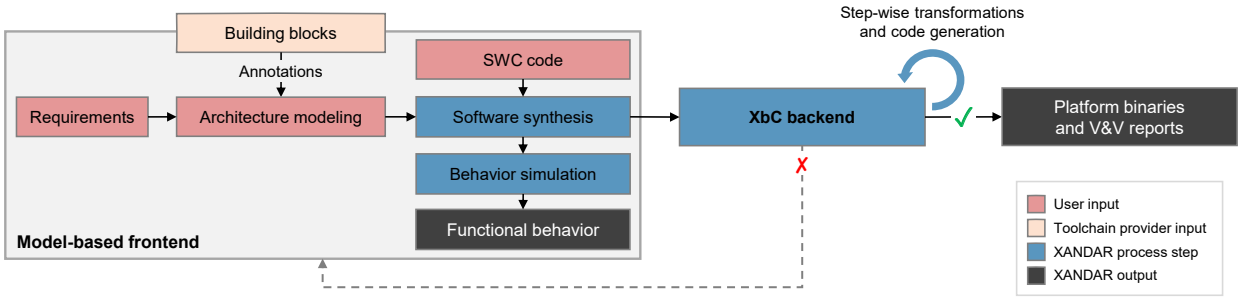

Fig. 2. XANDAR development process

hypervisor-based runtime architecture to complement this toolchain. XbC describes an automated process that refines a specification into a software system implementation and, during this refinement, ensures that certain non-functional properties are met by construction [3]. This technique has been applied to enforce confidentiality properties in software programs [4], for example. In the XANDAR project, the considered scope is shifted from individual programs to the entire embedded software system stack.

As shown in Figure 1, the XANDAR toolchain allows the developer to specify system requirements, model the architecture, and describe the behavior of individual applications using source code, primarily in the C programming language. Using a library of trusted building blocks, such as safety or security patterns, it generates an executable software system implementation in a semi-automatic manner. While doing so, it extracts relevant runtime properties and returns them as Verification & Validation (V&V) reports. Properties documented in these reports support developers in the V&V of functional and non-functional requirements.

A. Development Process

The toolchain allows developers to apply the XANDAR development process [5], which is shown in Figure 2 and can be decomposed into two main phases:

1) In the *model-based frontend*, the toolchain provides graphical and textual user interfaces to capture requirements, architecture models, and Software Component (SWC) code. As part of this process, models can be annotated with trusted building blocks.
2) In the *XbC backend*, step-wise transformations and code generation procedures are executed to create executable binaries for the target platform.

During this process, integrated analysis procedures identify relevant runtime properties and populate V&V reports. In addition, the fulfillment of selected requirements is automatically checked and can cause the process to abort. If this is the case, user inputs must be refined accordingly.

B. Organization of this Paper

The remainder of this paper is structured according to the development process. Section II first describes the modeling and software synthesis capabilities of the model-based frontend. Section III then presents the hypervisor-based runtime architecture developed by the XANDAR consortium, before Section IV describes how the XbC backend deploys designs to this architecture. Section V is dedicated to selected V&V capabilities of the toolchain. Finally, Section VI covers the evaluation of the toolchain using an automotive use case contributed by BMW and an avionics use case contributed by the German Aerospace Center (DLR).

II. Modeling and Software Synthesis

Inspired by PREEvision and its underlying metamodel [6], the toolchain captures requirements and the architecture across three abstraction layers: the logical function architecture, the software architecture, and the hardware architecture.

A. Architecture Modeling

The *logical function architecture* allows users to model behavior based on SysML activity diagram concepts, which are integrated into the architecture description language. In the toolchain, this capability has two use cases:

1) *Activity-driven system decomposition*: The hierarchical structure of the logical function architecture and necessary logical functions are derived.
2) *Detailed behavior description*: The behavior of individual logical functions is described and serves as a specification of functional requirements.

In the next step, the *software architecture* is derived and relevant *hardware properties* are specified. A prototype for these tasks was implemented in PREEvision.

As part of the architecture modeling procedure, users have the opportunity to specify additional timing requirements, as described in [7]. While modeling the logical function architecture, for instance, they can specify event chains and associated latency constraints; these specifications are used by the timing V&V toolset covered in Section V.

The software architecture can be exported to widespread exchange formats such as AUTOSAR [8]. In addition, it can be exported as an *XbC project model*, which is a custom textual notation that allows users to apply further refinements.

XbC project models describe the software architecture as a network of SWCs. In such a network, SWC ports are connected via unidirectional channels, while data consistency and

timing requirements are described using the Logical Execution Time (LET) paradigm [9]. It is possible to annotate model elements with trusted building blocks such as safety patterns, security patterns, or Artificial Intelligence (AI) backends. Patterns delegate the implementation of certain safety/security measures to later steps in the process; they are an integral part of XANDAR's cybersecurity strategy [10]. AI backends trigger readily available generators of ML inference code and make this code accessible from associated SWCs.

B. Software Synthesis

Based on an XbC project model, the toolchain guides users through an interactive code integration procedure. It generates *SWC code skeletons* and an Application Programming Interface (API) that allows developers to interact with the XANDAR runtime architecture. This software development methodology is further described in [11].

Based on integrated code, the model-based frontend performs a *software synthesis* process for each SWC. This process combines the user-provided code with auto-generated configurations and static libraries. The resulting artifacts are ready to be consumed by subsequent toolchain steps, such as the target deployment covered in Section IV or the behavior simulation framework described in Section V.

III. Runtime Architecture

The runtime architecture developed by the consortium is based on the XtratuM Next Generation (XNG) hypervisor and targets the following System-on-Chip (SoC) platforms:

- Xilinx Zynq-7000
- Xilinx Zynq UltraScale+ MPSoC
- Renesas R-Car V3U

XNG is a time- and space-partitioning type-1 hypervisor [12]. It was available for the Zynq-7000 and ported to the other two platforms as part of XANDAR. Among other guests, it supports bare-metal applications using the XtratuM Runtime Environment (XRE) and Linux. In the following, these guests are referred to as Runtime Environments (RTEs).

The developed runtime architecture extends its underlying RTE with XANDAR-specific features. Depending on the utilized RTE, it is available in two manifestations:

1) In the *XRE runtime*, every SWC is mapped to a dedicated XNG partition; inter-SWC communication is implemented using XNG channels.
2) In the *Linux runtime*, a single XNG partition hosts a Linux distribution, which in turn spawns a user-space process for every SWC; inter-SWC communication is implemented by a centralized orchestrator process.

Both runtimes are able to execute SWC networks as defined in XbC project models. They are able to enforce, for example, an LET-compliant communication between SWCs.

The runtime architecture is flexible with respect to manual extensions by the toolchain user, i.e., it is possible to add XNG partitions that are not managed by the XANDAR toolchain. It is further equipped with runtime features that

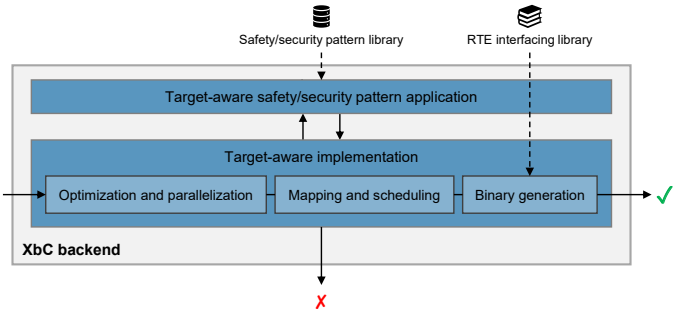

Fig. 3. Workflow implemented by the XbC backend

can be leveraged by annotated safety and security patterns. For example, a port logging functionality makes it possible to monitor and log all input and output events of SWCs, which can be an important feature to increase the transparency of decisions made by nondeterministic ML algorithms [13]. For the XRE runtime, a Runtime Security Monitoring (RSM) concept introduced in [14] detects and mitigates threats that emerge during the operation of a deployed software system, while the Health Monitor (HM) of XNG detects and confines other events that can compromise the system.

IV. Target Deployment and Optimization

The target deployment process is realized by the XbC backend and responsible for implementing a software system design (according to Section II) on the runtime architecture. Figure 3 shows a high-level overview of its workflow.

A. Optimization and Parallelization

The goal of the first XbC backend step is to apply an efficient combination of parameterized source code optimizations. Therefore, an automated procedure to extensively evaluate a large number of optimization parameters was implemented using PLUTO [15]. The procedure considers parallelization, loop tiling, register blocking, and loop merging/distribution properties. As part of this, a novel loop tiling methodology for single-threaded programs [16] and a register blocking technique for parallel loop kernels were developed.

B. Mapping and Scheduling

Following the optimization step, fully automated mapping and scheduling procedures are executed. These procedures depend on the targeted runtime architecture. For the XRE runtime, two alternative strategies were implemented:

1) The *greedy hypervisor deployment* schedules each activation of a SWC at the exact absolute time and for the exact duration of its LET time window [17].
2) The *optimizing hypervisor deployment* employs an Earliest Deadline First (EDF) scheduling scheme to derive a periodic schedule from specified LET constraints.

In both of these approaches, relevant Worst-Case Execution Time (WCET) bounds for user-provided SWC code must be provided to the XbC backend. In the greedy deployment, these bounds are used as a plausibility check, i.e., it is ensured

that accumulated WCETs of a SWC activation do not exceed the length of the associated LET time window. The approach gives developers maximum control over the devised hypervisor schedule, but it can lead to inefficient processor utilization. The optimizing hypervisor deployment uses LET parameters only to derive a data dependency graph; it pins environment interactions to their logical event times but leaves all other scheduling decisions to the EDF scheduler.

In the Linux case, activations of SWCs are explicitly managed by the orchestrator process. This process triggers a particular SWC binary, in general, whenever it is able to provide it with all required inputs. To make use of this feature, the XbC backend extracts relevant data dependencies from the specified set of LET parameters and feeds them to subsequent steps in the toolchain.

C. Target Binary Generation

The final task of the XbC backend is the creation of binaries to be executed on the target platform. To achieve this, all previously created artifacts (such as optimized source code or scheduling decisions) are combined with static libraries. By linking against the corresponding RTE interfacing library, for example, the user-provided code is automatically integrated into the underlying RTE (i.e., XRE or Linux).

In the XRE deployment, hypervisor configurations are automatically derived from the XbC project model and compiled using the XNG toolchain. The result of this process is a monolithic binary that is readily available to be deployed to the target hardware. In the Linux case, every SWC is compiled into a standalone binary. In addition, the orchestrator configuration is generated as an independent file. These artifacts can then be deployed to the targeted Linux distribution.

D. Safety/Security Pattern Application

As shown in Figure 3, the target-aware application of safety/security patterns is performed in parallel to the actual implementation procedure. Patterns can hook into the process and apply the required analysis, transformation, and generation steps. For patterns that are supported by runtime architecture features (cf. Section III), this involves the configuration of the underlying runtime feature. The aforementioned port logging pattern and the RSM concept are examples of this. However, the library does also contain *by-design patterns*, which do not necessarily generate runtime artifacts. An example of this is the Information Flow Control (IFC) safety pattern [18]. This pattern allows the user to annotate SWC ports with required and provided integrity levels. Based on this specification, it ensures that sufficient isolation measures prevent less trusted SWC outputs from interfering with safety-critical sinks, both internally and in the environment.

V. Verification and Validation (V&V)

To complement the safety/security pattern concept, the project partners developed a dedicated V&V toolset. Components from this toolset are often interactive and support developers to verify and validate properties that are not addressed by the building block library.

A. Static Code Analysis

The model-based frontend employs Polyspace Bug Finder and Polyspace Code Prover [19] to analyze SWC code with respect to defects, vulnerabilities, and inconsistencies.

Polyspace Bug Finder employs static analysis techniques to uncover runtime errors, concurrency issues, security vulnerabilities, and other defects. Polyspace Code Prover tackles more intricate issues like integer overflow, divide-by-zero, and out-of-bounds array accesses.

In the XANDAR toolchain, these static code analysis workflows are integrated into a Continuous Integration (CI) environment based on Jenkins. Toolchain users can leverage this integration to ensure that user-provided C and C++ code is continuously checked for issues.

B. Behavior Simulation Framework

A key component of the model-based frontend is its behavior simulation framework [11]. This interactive tool extracts the functional behavior from a software architecture model and synthesized SWCs. Based on LET parameters annotated to the software architecture, it is able to capture and report the logical timing of interactions between SWCs.

The results of this procedure allow users to verify that the specified behavior meets relevant functional requirements and to validate that the fulfillment of these requirements leads to the expected functionality.

This feature has Model-in-the-Loop (MiL) support, i.e., it allows users to co-simulate the system under design with an executable model of the envisaged environment. To achieve this, the simulator builds up on the Ptolemy II framework [20], which is fully wrapped by a command-line utility referred to as *XbCgen/sim*. The tool spawns a *simulation binary* of every SWC as a process of the host operating system. Afterwards, it initializes a discrete-event simulation in Ptolemy II, where every SWC is represented by an actor that connects to is simulation binary via a Unix domain socket. The output of such a simulation is a timed execution trace that captures the exact input-output behavior of every SWC.

C. Timing Verification Toolset

As described in Section II, the toolchain allows users to annotate timing requirements at different levels of granularity. The timing V&V toolset analyzes these specifications at different stages in the development process [7]. An analysis of the logical function architecture raises an error for inconsistent latency constraints, for example. This functionality is integrated into the model-based frontend. Other analyses require target knowledge and are therefore part of the XbC backend.

A particular example of a target-specific timing V&V capability is the memory interference simulation applied to the greedy hypervisor deployment. For this deployment strategy, the XbC backend is able to generate an AMALTHEA [21] model of relevant memory access events in the derived schedule. A simulation of these events can then identify timing anomalies caused by resource sharing effects.

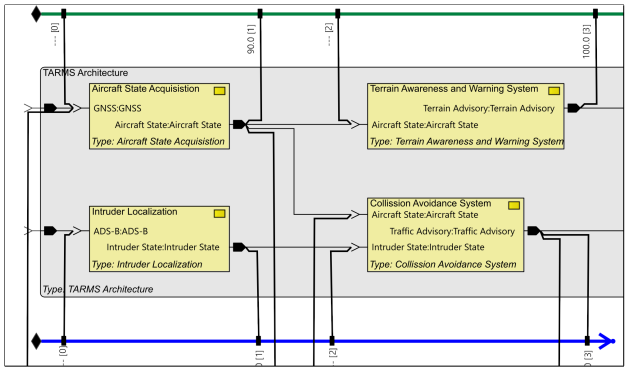

Fig. 4. Logical function architecture excerpt of the avionics use case

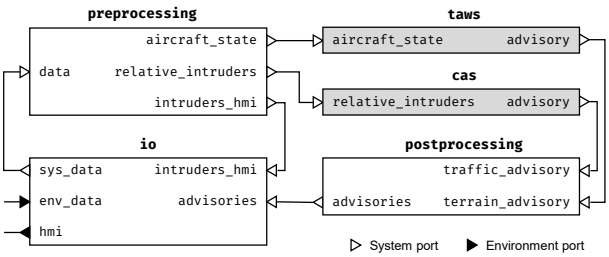

Fig. 5. Software architecture of the TARMS

VI. Demonstration and Evaluation

The development of the XANDAR design framework was guided by industrial partners from the automotive and avionics domain. Use cases contributed by them were finally investigated to evaluate benefits and limitations of the toolchain.

A. Avionics Use Case

As part of XANDAR, the German Aerospace Center (DLR) applied the toolchain to develop a Tactical Air Risk Mitigation System (TARMS) for the urban air mobility sector. In this field, air traffic density and ground proximity pose significant risks. Based on the own aircraft state and incoming transponder data of intruder aircraft, the TARMS provides pilots with collision warnings and terrain advisories.

Therefore, the modeling approaches from Section II were used to specify high-level behavior, apply an activity-driven system decomposition, model the detailed behavior, and annotate timing requirements. Figure 4 shows the decomposed system with annotated event chains. Afterwards, the software architecture in Figure 5 was derived, the consistency of timing specifications was checked, and SWC code skeletons were generated. The skeletons were then manually populated with application code written in both C and Rust. Due to its usage of a nondeterministic ML algorithm, the Collision Avoidance System (CAS) component was annotated with the port logging pattern provided as a building block.

Afterwards, the behavior simulation framework was used to verify and validate the behavior that the TARMS exhibits in response to flight scenarios such as the one shown in Figure 6. The highlighted CAS output shown in Figure 7, for example,

Fig. 6. FlightGear scenario used to test the CAS implementation

```
taws.main(15800..16000ms)
 - advisory = pb(080910011a020801)
io.main(16000..16200ms)
 - hmi_data = pb(0a020801121e0dd3b6684515000000001d0000000025000034432d0e40c9b
 - sys_sim_data = pb(0a1e0d41f57f42153333b5c11def819b42250000fa432d00000000350
preprocessing.main(16200..16400ms)
 - aircraft_states = pb(0d41f57f42153333b5c11def819b42250000fa432d000000003500
 - intr_data_hmi = pb(0dd17e2b3e15000000001d0000000025000034432d5e41c9bf359142
 - intruders = pb(0a280d9c667e44150000000001d0000000025000000002d5e41c9bf351073
postprocessing.main(16400..16600ms)
 - final_advisories = pb(0a1008081205088892c0081a05089a84a008120a080910011a040
cas.main(16600..16800ms)
 - advisory = pb(0808120208011a020803)
taws.main(16800..17000ms)
 - advisory = pb(080910011a020801)
```

Fig. 7. CAS advisory in the execution trace of a traffic scenario

encodes an advisory to descent, which is correctly issued in response to an imminent collision.

Selected cybersecurity considerations made during the design of the TARMS can be found in [22].

Finally, the toolchain was used to deploy the design to an avionics computer based on the Xilinx Zynq-7000. The hypervisor-based deployment strategy was chosen to provide the TARMS implementation with required real-time properties. As part of the deployment, target-specific timing V&V techniques were used to ensure that resource sharing effects do not cause critical deadlines to be missed.

B. Automotive Use Case

In the automotive context, BMW applied the toolchain to develop an environment perception system for automated and autonomous road vehicles. Based on camera and lidar data, this system implements sensor fusion functions to obtain a reliable model of the vehicle's surroundings.

Following the XANDAR development process, the completed steps are generally comparable to those conducted for the avionics use case. However, the detailed application of toolchain capabilities differs from that in the avionics use case.

The sensor fusion system applies the IFC pattern to ensure that camera inputs are always verified using lidar data before they are used to derive safety-related decisions. Furthermore, a secure onboarding pattern was applied to ensure that the runtime verifies the authenticity of each SWC before executing it. Further cybersecurity considerations made during this design process can be found in [23].

Since the addressed sensor fusion application depends on selected Linux drivers, the target deployment was conducted using the Linux runtime architecture of XANDAR. The resulting binaries were finally deployed to a Linux partition running on the Renesas R-Car V3U platform.

C. Evaluation Results

With respect to architecture modeling, the XbC building block library was found to relieve the developer of various manual tasks, which can increase productivity and reduce the risk of hazardous design errors. The decision to apply a certain building block, however, still has to be made by the developer. The automatic selection of patterns to apply for a given requirement set is currently not supported.

Regarding the software implementation step, the interactive code integration process was shown to be sufficiently flexible to support all use case features. However, driver code to access peripherals at the system boundary, such as an Ethernet controller, had to be integrated manually. Here, the automatic integration of target-specific device drivers would be beneficial to reduce development effort further.

With respect to V&V and target deployment, functional behavior and timing simulations proved helpful to detect systematic errors in the design. Combined with automatic target deployment mechanisms, they are able to improve the achieved software quality by a significant degree. However, the set of currently supported target platforms is limited and the automated portion of XANDAR's deployment strategies target either the XNG hypervisor or a Linux distribution. An automatic deployment of a SWC network to more than one such runtime platforms is currently not possible.

VII. Conclusion

XANDAR had the goal to provide a framework that supports developers in the creation of embedded software systems with certain safety, security, and real-time properties.

The use cases considered in the project are examples of modern and upcoming technology with a significant impact on the mobility domain. Furthermore, the technical requirements addressed by these use cases (e.g., high-performance hardware platforms, application of nondeterministic ML algorithms, or significant external connectivity) are important challenges in various other domains. The developed XANDAR framework was shown to be capable of addressing these requirements. It can therefore increase both productivity and the robustness of modern embedded software systems.

At the same time, the project showed that the vast complexity of autonomous and distributed embedded systems cannot be handled by a fully automated one-size-fits-all approach. As described above, the capabilities of the design framework had to be deliberately limited in various aspects. Code to access drivers for external communication, for example, currently has to be provided by the toolchain user.

XANDAR demonstrated the overall applicability of the XbC paradigm to various layers of the embedded software system stack. Developed concepts and tools interact to automate various aspects of the development process, but they do not eliminate the need for manual design steps.

The extensible nature of the XANDAR design framework (e.g., of the model-based pattern library) provides the opportunity to increase the achieved degree of automation even further. To explore this potential is future work.

References

[1] R. Ernst, "Automated Driving: The Cyber-Physical Perspective," *Computer*, vol. 51, no. 9, pp. 76–79, Sep. 2018.

[2] European Commission, "Fact sheet: X-by-Construction Design framework for Engineering Autonomous & Distributed Real-time Embedded Software Systems," https://doi.org/10.3030/957210.

[3] M. H. ter Beek, L. Cleophas, I. Schaefer, and B. W. Watson, "X-by-Construction," in *Leveraging Applications of Formal Methods, Verification and Validation: Modeling*, T. Margaria and B. Steffen, Eds. Springer International Publishing, 2018, vol. 11244.

[4] I. Schaefer, T. Runge, A. Knüppel, L. Cleophas, D. Kourie, and B. W. Watson, "Towards Confidentiality-by-Construction," in *Leveraging Applications of Formal Methods, Verification and Validation: Modeling*, T. Margaria and B. Steffen, Eds., 2018, vol. 11244.

[5] L. Masing, T. Dörr, F. Schade, J. Becker *et al.*, "XANDAR: Exploiting the X-by-Construction Paradigm in Model-based Development of Safety-critical Systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE '22)*, Antwerp, Belgium, Mar. 2022.

[6] J. Schäuffele, "E/E Architectural Design and Optimization using PREEvision," in *SAE 2016 World Congress and Exhibition*, Apr. 2016.

[7] R. Weber, N. Adler, T. Wilhelm, A. Sailer, and C. Reichmann, "Towards Automating a Software-Centered Development Process that considers Timing Properties," in *IEEE 35th International System-on-Chip Conference (SOCC)*, Belfast, United Kingdom, Sep. 2022.

[8] AUTomotive Open System ARchitecture (AUTOSAR). [Online]. Available: https://www.autosar.org/standards/classic-platform

[9] C. M. Kirsch and A. Sokolova, "The Logical Execution Time Paradigm," in *Advances in Real-Time Systems*, S. Chakraborty and J. Eberspächer, Eds. Springer, Berlin, Heidelberg, 2012, pp. 103–120.

[10] F. Siddiqui, R. Khan, S. Sezer *et al.*, "XANDAR: A holistic Cybersecurity Engineering Process for Safety-critical and Cyber-physical Systems," in *IEEE 95th Vehicular Technology Conference (VTC2022-Spring)*, Helsinki, Finland, Jun. 2022.

[11] T. Dörr, F. Schade, A. Ahlbrecht *et al.*, "A Behavior Specification and Simulation Methodology for Embedded Real-Time Software," in *26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, Alès, France, Sep. 2022.

[12] M. Masmano, I. Ripoll, and A. Crespo, "XtratuM: a Hypervisor for Safety Critical Embedded Systems," in *11th Real-Time Linux Workshop*, Dresden, Germany, 2009.

[13] European Union Aviation Safety Agency, "EASA Concept Paper: First usable guidance for Level 1 machine learning applications," Apr. 2021.

[14] H. Hui, K. McLaughlin, F. Siddiqui, S. Sezer, S. Y. Tasdemir, and B. Sonigara, "A Runtime Security Monitoring Architecture for Embedded Hypervisors," in *IEEE 36th International System-on-Chip Conference (SOCC)*, Santa Clara, USA, Sep. 2023.

[15] PLUTO - An automatic parallelizer and locality optimizer for affine loop nests. [Online]. Available: https://pluto-compiler.sourceforge.net

[16] V. Kelefouras, K. Djemame, G. Keramidas, and N. Voros, "A Methodology for Efficient Tile Size Selection for Affine Loop Kernels," *International Journal of Parallel Programming*, vol. 50, Aug. 2022.

[17] F. Schade, T. Dörr, A. Ahlbrecht, V. Janson, U. Durak, and J. Becker, "Automatic Deployment of Embedded Real-time Software Systems to Hypervisor-managed Platforms," in *2023 26th Euromicro Conference on Digital System Design (DSD)*, Durres, Albania, Sep. 2023.

[18] T. Dörr, F. Schade, and J. Becker, "Pattern-Based Information Flow Control for Safety-Critical On-Chip Systems," in *Computer Safety, Reliability, and Security*, J. Guiochet, S. Tonetta, and F. Bitsch, Eds. Springer, Cham, 2023, vol. 14181.

[19] MathWorks. Polyspace: Making critical code safe and secure. [Online]. Available: https://www.mathworks.com/products/polyspace.html

[20] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[21] Eclipse Foundation, Inc. APP4MC: Application Platform Project for MultiCore. [Online]. Available: https://eclipse.dev/app4mc

[22] F. Siddiqui, A. Ahlbrecht, R. Khan *et al.*, "Cybersecurity Engineering: Bridging the Security Gaps in Avionics Architectures and DO-326A/ED-202A," in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, Barcelona, Spain, Oct. 2023.

[23] F. Siddiqui, R. Khan, S. Y. Tasdemir *et al.*, "Cybersecurity Engineering: Bridging the Security Gaps in Advanced Automotive Systems and ISO/SAE 21434," in *2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*, Florence, Italy, Jun. 2023.