X-by-Construction Design Framework for Engineering Autonomous and Distributed Real-time Embedded Software Systems

Tutorial: Simulation-based development of networked avionics systems using the XANDAR toolchain

Tobias Dörr¹, Florian Schade¹, Alexander Ahlbrecht²
¹Karlsruhe Institute of Technology (KIT)
²German Aerospace Center (DLR)

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

- **Part I:** The software development methodology of XANDAR
  - Motivation + XANDAR project overview
  - Software development approach
  - Simulation and target deployment
  - Safety pattern concept + two sample patterns

- **Part II:** Programming models for concurrent systems
  - Fundamentals + selected programming models
  - Applicability to the XANDAR toolchain

- **Part III:** Live demonstration of XbCgen
  - Overview and introductory examples
  - Application to an avionics use case
Part I

The software development methodology of XANDAR
Motivation

- Designing embedded systems is an error-prone task

  - Programming error in the control software [1]
  - Failure of the Inertial Reference System during test flight
  - Self-destruction after 37 seconds

Remote access to 1.4 million road vehicle (2015):
  - Vulnerability in the vehicle’s head unit [3]
  - Remote reprogramming of a gateway component
  - Full access to a CAN bus of the vehicle

Motivation

- **Properties of modern cyber-physical systems** aggravate the design challenge:
  - Consolidation of functions on multicore platforms, e.g. in the automotive domain
  - Functional safety and cybersecurity requirements, e.g. fault tolerance
  - Integration of Artificial Intelligence (AI) algorithms, e.g. for object detection
  - Continuous interaction with remote entities, e.g. in a cloud-edge setting

- **Conflicting requirements** are particularly difficult to address, for example:
  - **Centralization of automotive on-board architectures** [4]
    - Lack of physical separation between components
    - Potential for timing interferences, fault propagation, ...
    - Functional safety issues

Motivation

- **Methodological gap** between requirements and implementations, e.g.:
  - ISO 26262 [5] specifies Freedom from Interference (FFI) requirements
  - A hypervisor such as XtratuM [6] is a building block that contributes to this goal
  - The correct application of such building blocks can be difficult (timing interferences, ...)

**ISO 26262** — Functional safety standard for road vehicles (initially published in 2011). Covers the following scope [5]: “This document addresses possible hazards caused by malfunctioning behaviour of safety-related E/E systems, including interaction of these systems.”

**Hypervisor** — Software layer providing independent execution environments on a single processor. Hypervisors running directly on the target hardware (without a host OS) are called type-1 hypervisors.

**Motivation**

▪ **Motivation:** How to apply such building blocks in a provably correct manner?

Excerpt of ISO 26262-11:2018

5.4.2.2 Clarifications on Freedom from interference (FFI) in multi-core components

If in a multi-core context multiple software elements with different ASIL ratings coexist, a freedom from interference analysis according to ISO 26262-9:2018, Clause 6 is carried out.

“Techniques such as hypervisors can help to achieve software partitioning […]”

### X-by-Construction (XbC)

- The “step-wise refinement process from specification to code that automatically generates software (system) implementations that by construction satisfy specific non-functional properties concerning security, dependability, reliability, or resource/energy consumption, to name but a few” [7].

**Key idea:** Apply the X-by-Construction (XbC) paradigm to the design process

▪ Auto-generate implementation artefacts that leverage suitable low-level techniques

▪ Simplify the post-hoc verification process in a manner comparable to [8]


Overview of the XANDAR project

**Goal:** Deliver a mature software toolchain that uses the X-by-Construction paradigm to generate system implementations with guaranteed properties

**Duration:**
01/2021 – 12/2023

**Budget:**
€ 4.96 million

**Project coordinator:**
Prof. Jürgen Becker (KIT)

**Scientific coordinator:**
Prof. Nikolaos Voros (UoP)

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

- The toolchain allows users to apply the XANDAR development process [9]
- This process defines a sequence of **iteratively traversed steps**:

Acronyms: XbC = X-by-Construction, SWC = Software Component, HW = Hardware

Overview of the XANDAR project

- With every iteration, the architecture model is refined or extended
- Relevant abstraction levels inspired by the PREEvision [10] layer model

**Evolution of an architecture model** along the development process:

Overview of the XANDAR project

- **Toolchain evaluated** in a laboratory setup with two use cases:
  - DLR ⇒ Resilient Avionic Architecture for Urban Air Mobility (UAM)
  - BMW ⇒ Autonomous Systems with Integrated Machine Learning Applications

- **Logical architecture** excerpt of DLR’s pilot assistance system for UAM [11]:

![Diagram of Tactical Air Risk Mitigation System (TARMS)]


Software development approach

- **Description of the software architecture** by the toolchain user:
  - Software Component (SWC) entities communicating via message passing
  - Logical Execution Time (LET) parameters [13] capture the desired timing behaviour

- **Software architecture metamodel** as class diagram:

Software development approach

- **Sample network of SWCs** and textual description in JSON5 [14] syntax:

```json
acquisition: {
    period: 10,
    activations: {
        check: {
            offset: 0,
            runtime: 3,
            write_to: [ 'alive' ],
        },
        gen: {
            offset: 3,
            runtime: 7,
            write_to: [ 'data' ],
        },
    },
    system_ports: {
        data: { /* ... */ },
        alive: { /* ... */ },
    },
    env_ports: {
        val: { /* ... */ },
    }
}
```

- **LET frame sequence:**

Software development approach

**Logical Execution Time (LET)** – “LET determines the time it takes from reading program input to writing program output regardless of the time it takes to execute the program” [15].

- **Generation** of SWC code skeletons:

  ```c
  // Initialization hook:
  void swc_init(void);
  
  // Trigger hook:
  void swc_trigger(enum activation_id activation, struct swc_port_map *port);
  ```

- **Access to input/output ports** via the framework-provided port map
  - Support for structured data structures (based on the Protocol Buffers [16] format)
  - Built-in support for access to input/output controllers of the hardware

- **On the target hardware:** invocation of the trigger hook for each SWC activation


Simulation and target deployment

- **Starting point**: Fully deterministic specification of the software behaviour
- **Tool support** provided by the XbCgen framework of XANDAR:

```
Starting point: Fully deterministic specification of the software behaviour

XbCgen/sync → Skeleton generation → Code integration → XbC backend

XbCgen/sim → Behaviour simulation → XbCgen/deploy

```

7 June 2023
Simulation-based development of networked avionics systems using the XANDAR toolchain
Simulation and target deployment

- **XbCgen/sim**: Generation of execution traces for verification and validation
  - Discrete-event simulation with a user-provided plant/environment model (in Ptolemy II)
  - Automatic synthesis of an executable model (calling SWC simulation binaries)
  - Refer to previous work in [12] for a detailed description

- Sample excerpt of an execution trace:

```
acquisition.gen(3.10ms)
  - data = [1566, 1564, 1563]
acquisition.check(10.13ms)
  - alive = true
monitor.main(13.15ms)
  - res = 1564.3333333333333
processing.main(10.20ms)
  - res = 1564.3333333333333
acquisition.gen(13.20ms)
  - data = [1558, 1557, 1555]
acquisition.check(20.23ms)
  - alive = true
monitor.main(23.25ms)
```

**Simulation and target deployment**

- **XbCgen/deploy**: Automatic generation of implementation artefacts for MPSoCs

  - The target runtime environment consists of:
    - A hypervisor layer currently implemented using the XtratuM hypervisor
    - An optional operating system layer (RTEMS, FreeRTOS, Linux, ...)

- SWCs are deployed to XtratuM partitions or OS tasks/processes, for example:

```
  +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+
  |     SWC_1     |   |     SWC_2     |   |     SWC_3     |   |     SWC_4     |   |     SWC_5     |   |     SWC_6     |   |     SWC_7     |
  +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+
  | RTEMS          |   |   XtratuM hypervisor     |   |   XtratuM hypervisor     |   |   XtratuM hypervisor     |
  +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+
  | Core cluster   |   | Core cluster   |   | Core cluster   |   | Core cluster   |
  +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+
  | MPSoC           |   | MPSoC           |   | MPSoC           |   | MPSoC           |
  +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+     +----------------+```
Safety pattern concept

- **Xbc pattern library**: Collection of verified design-time procedures for safety and security that can be annotated to system entities [17]

- Automatic implementation of the corresponding mechanisms:

Safety pattern concept

- **Pattern usage**: Explicit specification in the textual model instance (JSON5)
- **Example**: Triple Modular Redundancy (TMR) for fault tolerance

```json
annotations: {
    proc_redundancy: {
        id: 'software_tmr_pattern',
        target: 'processing',
    },
}
```

- **Next up**: Information Flow Control (IFC) + runtime logging for AI components

Generation of redundant partitions and an ‘arbitrator sandbox’ in the sense of [18]

**Motivation**: Unintended information flow via shared on-chip resources can lead to the violation of safety/security requirements, for example...

**Concept**: Auto-generate APU configurations that are as prohibitive as possible and compare potentially feasible information flows to an accept list.

Image source: [19]

**Information Flow Control (IFC) safety pattern**

- **Underlying methodology** from the state of the art [20]:
  - Inputs are the system architecture and the desired information flow policy ($I_A$)
  - Graph-based algorithm to determine all potentially feasible information flows ($I_F$)

![Algorithm 1: $G_B$ construction from a model instance](image)

**Xbc guarantee:** $I_F \subseteq I_A$ holds for interactions between SWCs, ports, ...

Runtime logging for AI components

- Upcoming normative and legislative constraints on AI components:
  - Ethics of Connected and Automated Vehicles [21] by the European Commission
  - EASA Concept Paper: First usable guidance for Level 1 machine learning applications [22]

- Selected excerpts focused on the transparency of algorithmic decisions:

  “User-centred methods and interfaces for the explainability of AI-based forms of CAV decision-making should be developed.”
  Source: [21]

  “Did you put adequate logging practices in place to record the decision(s) or recommendation(s) of the AI-based system?”
  Source: [22]

Runtime logging for AI components

- **Library-provided logging support** at the level of SWC ports:

  - **XbC guarantee**: Correct synthesis and deployment of logging modules
    - Allocation of a dedicated memory region (incl. IFC pattern compatibility)
    - Non-interference with the LET-based communication model
Part II
Programming models for concurrent systems
Fundamentals

**Concurrency** – Two or more actions are in progress at the same time [23].

**Parallelism** – Two or more actions execute simultaneously [23].

**Shared memory communication** – Concurrent modules communicate via objects in memory [24].

**Message passing** – Concurrent modules communicate by transmitting messages over a channel [24].

- Cyber-physical systems (CPS) are inherently concurrent [25]
  ⇒ Approaches to achieve deterministic concurrency are essential

- Incorrect data sharing is a common source of concurrency issues [26]
  ⇒ Techniques based on message passing particularly interesting for CPS applications

---


Kahn Process Networks (KPN)

- **Idea:** simple language for parallel programming [27]
  - Processes (computing stations) = sequential programs with Algol-like syntax
  - Channels (communication lines) = unbounded first-in-first-out queues
  - Asynchronous message passing (nonblocking writes and blocking reads)

- KPN programs are deterministic [28]:
  
  "Kahn showed that concurrent execution was possible without nondeterminism"

- Expressivity deliberately limited

---

Communicating Sequential Processes (CSP)

- **Key principle:** parallel composition of sequential processes [29]
  - Introduced to coordinate/synchronise multiprocessor machine communication
  - Synchronous message passing (blocking writes and blocking reads)

- Part of programming languages such as occam-\(\pi\) [30] and Go [31]:

```go
func main() {
    exit := make(chan bool)
    channel := make(chan uint8)
    go consumer(channel, exit)
    go producer(channel)
    <-exit
}
```

```go
func consumer(in <-chan uint8, exit chan<- bool) {
    fmt.Println(<-in)
    exit <- true
}
```

```go
func producer(out chan<- uint8) {
    time.Sleep(time.Second)
    out <- 42
}
```

Synchronous languages

- **Key principle**: combination of synchrony and concurrency [32]
  - Examples are Lustre [33] or more recent approaches such as Blech [34]
  - Immediate reactions (simultaneous reads and writes)
  - Temporal semantics based on a global clock

- Strong foundation for formal verification

- Widespread use for safety-critical system design:
  - Lustre is a fundamental part of the SCADE toolset by Esterel Technologies
  - Applications include nuclear power plants and flight control software [32]

---

[34] F. Gretz, F.-J. Grosch, “Blech, Imperative Synchronous Programming!”, FDL '18, September 2018.
Logical Execution Time (LET)

- **Origin**: Giotto methodology for embedded control system design [13]
  - Periodic invocation of sequential tasks ⇒ time-triggered framework
  - Reading of input ports at the invocation time
  - Writing of output ports at the end of the period

- **Other manifestations of the LET abstraction:**
  - xGiotto for event-driven programming [35]
  - Timing Definition Language (TDL), cf. [36]
  - System-level LET for distributed systems [37]
  - The synchronous LET paradigm [38]

---

Reactors and Lingua Franca (LF)

- **Concept**: reactors as coordination model for concurrent systems [39]
  - A reactor is composed of **reactions** in a target language (C, Python, Rust, ...)
  - Reactions are executed in response to **trigger events** (carrying a value and a tag)
- **Lingua Franca (LF)** for the development of reactor-based systems:

```c
#include <stdio.h>

int main()
{
    timer t(0, 1 sec);
    reaction(t) {
        printf("Timer expired!\n");
    }
    return 0;
}
```

“Lingua Franca is a polyglot coordination language for reactive, concurrent, and time-sensitive applications.”

www.lf-lang.org

Applicability to the XANDAR toolchain

- Compatibility with the full feature set of XANDAR is essential

- **LET has been shown to offer a promising trade-off**
  - Straightforward integration into the overall framework
  - Compatible with the requirements of XANDAR’s use cases

- Relaxation of current constraints is future work

- Ideas for steps beyond the current model:
  - Integration of LF, which is a generalisation of LET [40]
  - Add support for the synchronous LET paradigm [38]

---

Part III

Live demonstration of XbCgen
Overview and introductory examples

1. Software architecture modelling
2. SWC skeleton generation (XbCgen/sync)
3. SWC code development (in C)
4. Simulation design
5. Execution trace generation (XbCgen/sim)
Application to an avionics use case

- SWC chain from the Tactical Air Risk Mitigation System (TARMS) by DLR:

![SWC chain diagram]

Neural network implementation of horizontal and vertical CAS based on the approach from [41]

```java
enum VerticalAdvisory {
    COC = 1; // Clear of conflict
    DNC = 2; // Do not climb
    DND = 3; // Do not descend
    DES1500 = 4; // ...
    CL1500 = 5;
    SDES1500 = 6;
    SCL1500 = 7;
    SDES2500 = 8;
    SCL2500 = 9;
}
```

Application to an avionics use case

- Sample activation pattern specification:

- FlightGear [42] scenarios serve as simulation stimuli:

Closing

Summary
Summary

- **Software development methodology of the XANDAR toolchain:**
  - Software modelled as a network of SWCs with LET parameters
  - Pre-verified safety/security patterns are annotated by the toolchain user
  - Behaviour simulation + automatic deployment to a type-1 hypervisor on MPSoCs

- **Programming models for concurrent systems:**
  - LET has been shown to be a suitable model for the XANDAR framework
  - Relaxation of current constraints (e.g. by moving to LF) is future work

- **Live demonstration of XbCgen:**
  - Flow from modelling to execution trace generation
  - Application to the DLR use case