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

Research and Innovation Actions

Horizon 2020, Topic ICT-50-2020:
Software Technologies

Grant agreement ID: 957210

– Deliverable –

D3.1: Programming Model Specification

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

Document information

Document title:	Programming Model Specification
Work package:	WP3
Editor:	KIT
Author(s):	KIT, UOP, AVN, VECTOR, QUB, FENTISS
Reviewer(s):	VECTOR, BMW
Document type:	Report
Version:	1.0
Status:	Released
Dissemination level:	Public

XANDAR consortium

No.	Short name	Name	Country
1	KIT	Karlsruher Institut für Technologie	Germany
2	UOP	University of Peloponnese	Greece
3	DLR	Deutsches Zentrum für Luft- und Raumfahrt	Germany
4	AVN	AVN Innovative Technology Solutions Limited	Cyprus
5	VECTOR	Vector Informatik GmbH	Germany
6	BMW	Bayerische Motoren Werke Aktiengesellschaft	Germany
7	QUB	The Queen's University of Belfast	United Kingdom
8	FENTISS	Fent Innovative Software Solutions SL	Spain

Copyright & disclaimer

Document revision history

Version	Date	Comments
1.0	2021-06-29	Publication of the final version.

About this document

This document describes the XANDAR programming model, which lays the foundation to enable both the simulation-based verification of functional requirements and the realization of non-functional requirements (such as safety and security) by pattern-based model and code transformations. Both aspects are subject to research in the further course of the XANDAR project.

Section 1 describes the XANDAR design methodology, which is the procedure for developing software for embedded systems according to the XANDAR approach. It is based on the XANDAR process as described in deliverable 2.1.

Section 2 describes central aspects of target platforms and runtime environments to which software applications are deployed during the system generation of the XANDAR process.

Finally, section 3 describes the programming model that the XANDAR toolchain user, i.e. the developer of the system under consideration, is expected to follow during the software component implementation step of the XANDAR process. This enables further processing of the implementations in the XANDAR toolchain, both for simulation and for deployment on the target platform.

Table of contents

List of figures

Glossary of terms

API	Application Programming Interface
BSW	Basic Software
ECU	Electronic Control Unit
LIC	Low-Level Interfacing Component
LLDD	Low-Level Device Driver
LLDI	Low-Level Device Interface
OS	Operating System
PC	Processing Component
RTE	Runtime Environment
SWC	Software Component
WCET	Worst-Case Execution Time
XNG	XtratuM Next Generation

1 XANDAR design methodology

This document builds upon the XANDAR development process as outlined in deliverable 2.1 of the project. Focus is put on the procedure that developers, i.e. users of the developed toolchain, shall follow when designing software applications for an embedded system. This procedure is referred to as the "XANDAR design methodology" in the following and briefly introduced in this section. The next sections provide a more detailed description of several relevant aspects that are related to this methodology.

Both the overview that is provided in this section and the details as given in the following sections serve as a basic framework for future XANDAR activities. During the course of the project, this initial framework will be continuously refined to incorporate specific tools and approaches that are planned to be developed in upcoming tasks of WP 2 and WP 3.

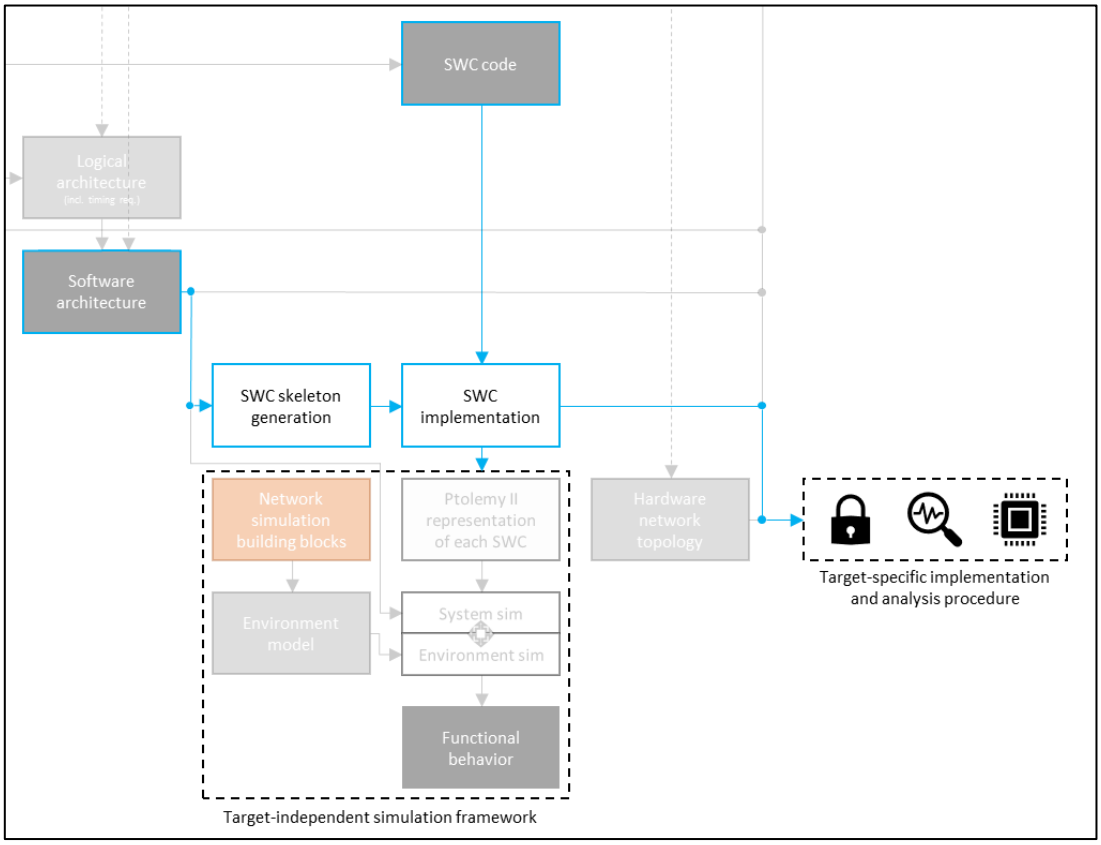

Figure 1-1: Excerpt from the XANDAR development process

Figure 1-1 shows a simplified excerpt from the XANDAR development process (see XANDAR deliverable 2.1). All input artefacts (grey) and process steps (white) that are directly related to the design methodology are highlighted using blue borders.

One of the input artefacts that the process expects to be provided by the developer is the software architecture. The basic building blocks that are used to construct such a software architecture are called software components (SWCs). In the XANDAR design methodology, a SWC has a specified set of input ports as well as a specified set of output ports. These ports are used to interconnect the instantiated SWCs. More specifically, a directed connection from an output port of SWC_A to an input port of SWC_B corresponds to a message that SWC_A transmits to SWC_B. Furthermore, SWCs can also be connected to specialized hardware

components such as sensors, actuators, or explicitly managed external memory. This type of SWCs possesses dedicated ports to interact with such specialized components. The following categorization lists the two possible types of SWCs:

1. **Processing component (PC):** A kind of SWC that only interacts with other SWCs using input and output ports.
2. **Low-level interfacing component (LIC):** A kind of SWC that combines all capabilities of a PC with the ability to exchange messages with low-level devices (such as an on-chip I/O controller used to interact with an external sensor) using bidirectional ports.

The external interfaces of a PC are visualized in Figure 1-2; Figure 1-3 shows the external interfaces of a LIC.

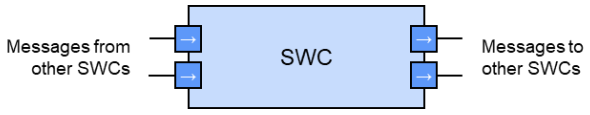

Figure 1-2: External interfaces of a processing component (PC)

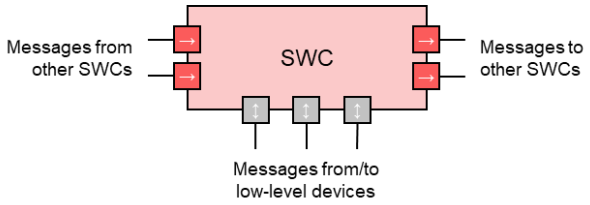

Figure 1-3: External interfaces of a low-level interfacing component (LIC)

From the perspective of the design methodology, a software architecture is an input artefact that is defined by the developer through a specification of SWCs and their interconnections. An example software architecture that consists of two PCs and three LICs is shown in Figure 1-4. Note that in this figure, input and output ports of PCs are connected exclusively to other PCs or LICs. For each LIC, however, a connection to a specialized hardware component (a sensor, an actuator, or a non-volatile memory component) is specified.

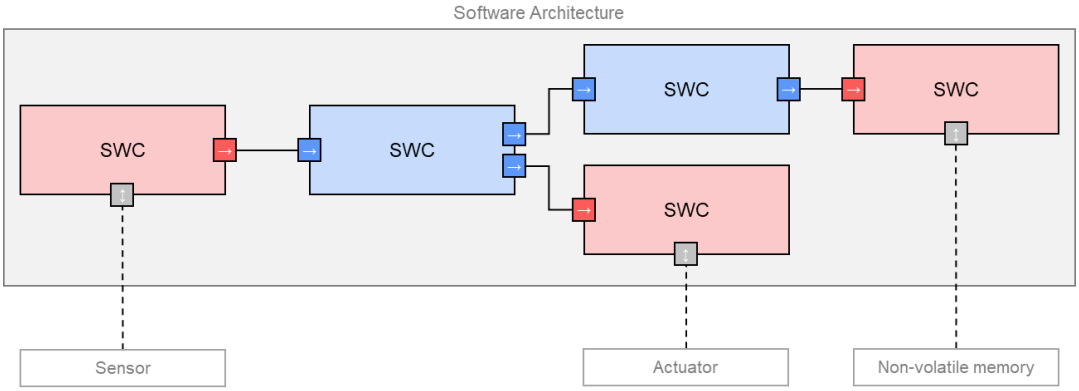

Figure 1-4: Example of a software architecture with two PCs and three LICs

As shown in Figure 1-1, the software architecture first passes a processing step that generates a “skeleton” for every SWC. This skeleton provides mechanisms that enable the developer to interact with the ports of the SWC. It serves as an abstraction of the actual target that the SWC will later be executed.

In order to populate the SWC skeleton, the developer creates SWC code and integrates it into the corresponding skeleton (Figure 1-5). This can be performed manually by writing code that implements the desired behaviour. However, the proposed approach is flexible in the sense that available means to generate code (such as generated code from MATLAB/Simulink models) can be used as an alternative. Furthermore, code can also originate from generation facilities of AI/ML frameworks or from previous versions of a developed system in form of legacy code.

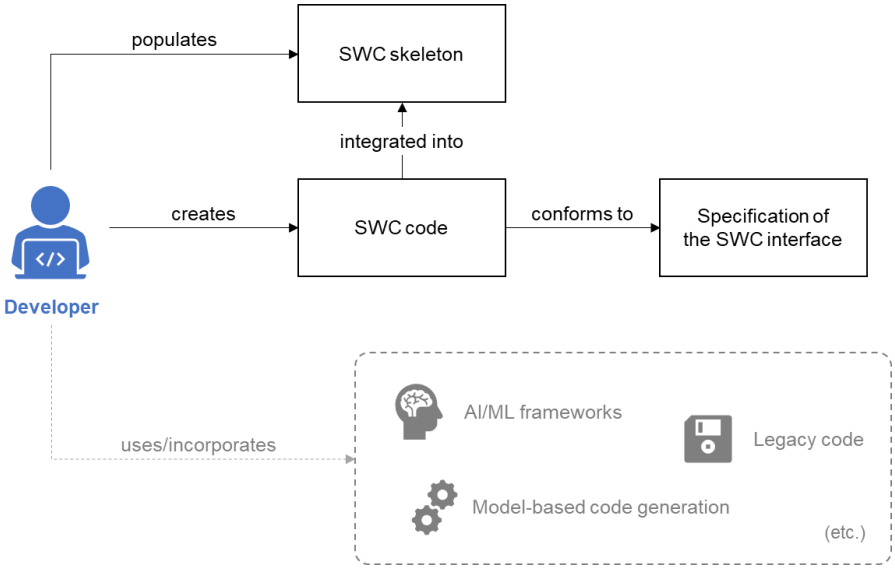

Figure 1-5: SWC code creation procedure

In any case, the code that is integrated into the SWC skeleton must conform with the SWC interface specification. For the purposes of the project, we refer to the model that describes this interface as the “SWC programming model”. From an implementation perspective, it is given in the form of an API that the developer shall make use of. In other words: To populate a SWC skeleton, the developer shall comply with the interface as defined by the SWC programming model. In a specific programming language such as C, this is accomplished by strict adherence to the specified API.

As soon as code that meets the respective requirements is available for every SWC of the software architecture, the XANDAR toolchain performs a “SWC implementation” step in which every SWC is automatically adapted such that it can be passed to both the “target-independent simulation framework” and the “target-specific implementation and analysis procedure” (Figure 1-1). In the former case, every SWC must be wrapped into a Ptolemy II actor that can be used to simulate its behaviour. In the latter case, SWCs must be refined such that they can be executed on the target runtime environment (RTE).

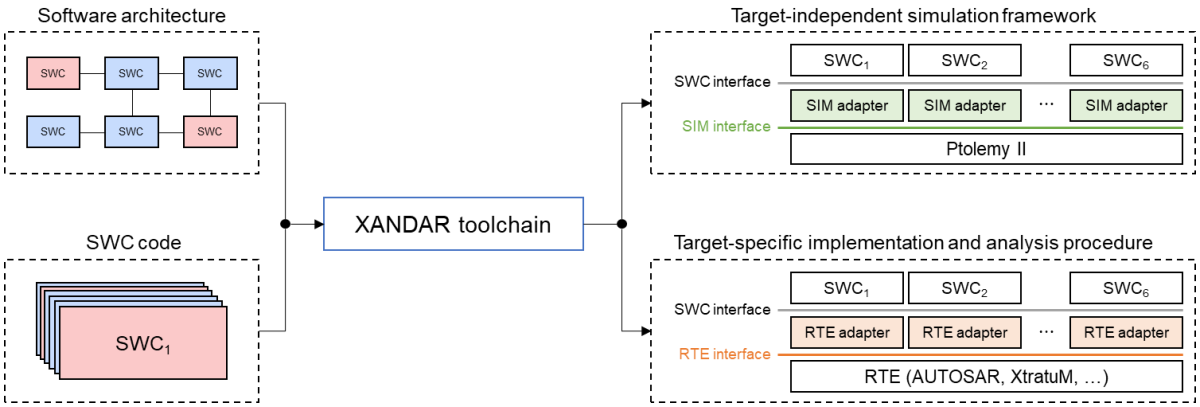

Figure 1-6: Generation of SIM and RTE adapters

Figure 1-6 visualizes the “SWC implementation” step in more detail: Starting from the provided software architecture and code for every SWC, a simulation adapter (“SIM adapter”) as well as a runtime environment adapter (“RTE adapter”) is generated for every SWC.

As illustrated in Figure 1-6, the “SWC interface” is the interface between every SWC and the adapters. It is described by the “SWC programming model” and developers must adhere to it while writing SWC code. Further interfaces involved in this process are the “SIM interface” and the “RTE interface”. While the former specifies how an SWC can be wrapped as a Ptolemy II actor, the latter is used to translate all calls that are made by a SWC into a form that can be executed by the target RTE.

This document specifies the RTE interface in section 2 and the SWC interface in section 3. Due to its close relationship with the functional verification, the SIM interface is beyond the scope of this document and covered in the WP5 deliverables.

2 Runtime programming model

2.1 Machine model

The runtime programming model defines a set of features that the RTE provides to the XANDAR toolchain. These features depend on the target platform, which can be generalized into an abstract machine model. This machine model defines the elements of the underlying physical hardware, its features, and the scope that is supported within XANDAR. It provides a generalization among all potential target hardware architectures and describes them in a unified way. While the described machine model may be restrictive, the XANDAR approach can easily be extended and adapted to a wider machine model.

A hardware platform in XANDAR consists of processing elements, memory elements, I/O elements, interconnect elements, and other hardware elements. The processing elements may refer to either single-core, multi-core, or special processing units. Special processing units are restricted to memory-mapped accelerators. In the context of XANDAR, they specifically refer to GPU or AI accelerators. The memory consists of caches, a shared main memory, and local scratchpad memory. I/O refers to communication interfaces that allow data exchange with any kind of connected device, e.g. sensors, other controllers, or a cloud infrastructure. A hardware platform may also provide additional hardware elements such as memory management units (MMU) as needed for safety and security features for example.

XANDAR targets high-performance multi-core SoCs such as the Renesas R-Car or NVIDIA Xavier series. Those platforms provide the necessary processing power to deal with increasing computationally expensive high-level software tasks as common in AI for example. At the same time, XANDAR strives to support classic microcontrollers such as the AURIX TriCore. The latter is widely used in projects with hard real-time constraints. Such use cases may impose additional restrictions on the basic machine model, especially in regard to predictability, i.e. allowing worst-case execution time (WCET) analysis tools to calculate clear upper bounds.

2.2 AUTOSAR Classic & AUTOSAR Adaptive

AUTOSAR (classic and adaptive) provides a rather generic methodology to model the system and its interfaces (in AUTOSAR modelled as ports of software components). An abstract illustration of the AUTOSAR classic methodology is given in Figure 2-1. Figure 2-2 gives a similar overview for AUTOSAR adaptive.

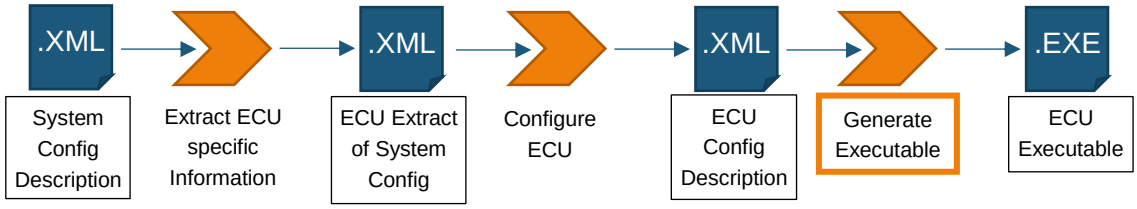

Figure 2-1: Outline of the AUTOSAR classic methodology

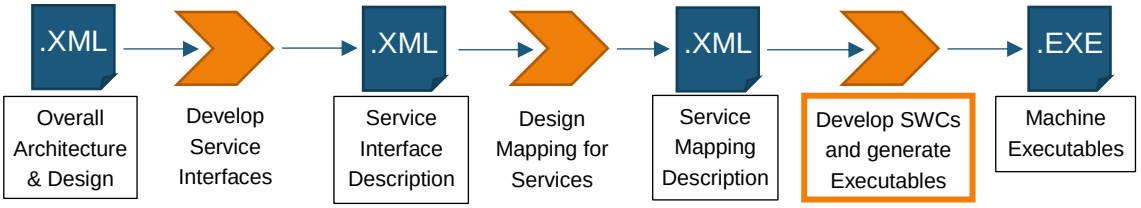

Figure 2-2: Outline of the AUTOSAR adaptive methodology

For a more detailed description of both methodologies, refer to [1] and [2]. Starting in the middle of the methodology in Figure 2-1, after configuring the ECU we get the ECU configuration description as a result (see [3] for details). This description contains all ECU-specific information that is necessary to generate the structural code that provides the APIs, the behavioural code can interact with. This set of information includes for example scheduling configuration, necessary basic software (BSW) modules, low level drivers for the hardware, BSW configurations, assignment of runnable entities to OS tasks, provided and required local/external communication interfaces, etc. Figure 2-2 shows the relevant steps to generate service communication for adaptive AUTOSAR: first, the service interfaces are described; second, they are mapped to network connections of machines; and third, with the service interface and mapping description as well as the software implementation the machine executables are generated.

The step “Generate Executable” (highlighted in the white boxes with the orange border) suggests a somewhat simple code generation and compilation step. However, there are a number of things involved in this generation. To enable platform independence, the software components and their communication via ports rely on the runtime environment (RTE) in classic and service interface description/mapping in adaptive. They provide the communication interfaces (required and provided) that are directly derived from the ports, their connections, names, and interface types. Under the hood, they take care about how the data is transferred (e.g. local memory, shared memory for core-to-core communication, bus messages for inter ECU communication, etc.). For a simple required port “Signal_CalcVehSpeed” in a software component “CalcVehSpeed_Plaus” which is generated as a sender-receiver interface the RTE would for example generate a function with the following signature:

```
void Rte_Read_Signal_CalcVehSpeed_Plaus_Signal_CalcVehSpeed(Signal_CalcVehSpeed* vehSpd);
```

Obviously, there is a lot more to the code generation than this simple example can convey. Nonetheless, most importantly, the AUTOSAR standard only demands that the behavioural part of the code shall call the communication methods as generated by the RTE (cf. [4], section 3.4). It does not define how the RTE methods should be named, just that they have to be compatible and that a number of principles should be followed (refer to [4], section 5.2) – this is similar for adaptive systems. For this reason, the AUTOSAR OS (classic) and middleware (adaptive) vendors provide their own RTE and service interface generators with corresponding documentation. The RTE, for instance, generated by the Vector MICROSAR [5] RTE generator can therefore differ from the Elektrobit Tresos [6] RTE generator.

There is another aspect, relevant for WP3: In order to generate functional/behavioural code (the parts that *call* the RTE methods), we need to know which methods are provided by the RTE. Therefore, code generators for behaviour cooperate with the AUTOSAR OS/middleware

vendors to target their generated code specifically for the vendor-generated RTE. The Simulink AUTOSAR Blockset [7], for example, offers code generation for Vector MICROSAR. In turn, given the same ECU Configuration Description (.arxml), both the MICROSAR RTE generator and the Simulink AUTOSAR Blockset generate code, and both parts can be integrated (interfaces are correctly named, used, and implemented) and compiled.

2.3 XNG hypervisor

2.3.1 Temporal system model

The XtratuM Next Generation (XNG) hypervisor uses the system-wide end-to-end-flow model. In this model, tasks are the execution units (a thread, a function or a process), characterized only by the intrinsic properties of its code: computation time and the resources (critical sections) used. The timing constraints are associated to the end-to-end-flow and not to the tasks.

The elements of the system are:

Partition: A container that encapsulates a number of tasks. Tasks are directly scheduled by the partition itself.

Hypervisor: Enforces the spatial and temporal isolation among partitions.

Task: The elemental execution unit. Every task is assigned a partition and executed within the partition context. Tasks are characterized by their computation time. Tasks can be pre-empted.

Mutual-exclusion-resource: An abstract object, used to define mutual exclusion between tasks.

End-to-end-flow: A sequence of tasks with temporal attributes. The end-to-end-flow is the element that has periodic behaviour and defines the temporal constraints. The tasks that are part of an end-to-end-flow can belong to different partitions. That is, an end-to-end-flow is not linked with a specific partition, it is a system-wide element.

End-to-end-flows are used to define the temporal behaviour of the system.

Plan: Defines the set of end-to-end-flows, which characterize the workload of the system in different operation modes. For example, the user may define a plan for the initialization phase of the system, where only a few end-to-end-flows are active. A second plan may define the full operation mode, where all resources are active.

For each plan, the user will be able to use the configuration tool (introduced in the following section) to generate a valid allocation of CPU time to the partitions in the such a way that the temporal constraints of the system, as described by means of the rest of rest of the temporal model (end-to-end-flows, steps, mutual-exclusion-resources, …) are acknowledged.

2.3.2 Xoncrete tool

The configuration XNG is defined during the development stage. The representation of this static configuration is stored in the XCF (XNG configuration files) file.

Xoncrete is an analysis tool, which captures the system description, performs the schedulability and sensitivity analysis, and provides a report with the results. The specification of the temporal system model elements is the input of the tool, and it provides the XCF file, thereby the XNG hypervisor, with the XCF file as input, is configured. The following figures show the results of the schedulability analysis, the report with the slots provided for each partition at the execution time (Figure 2-3) and the CPU usage by each partition (Figure 2-4).

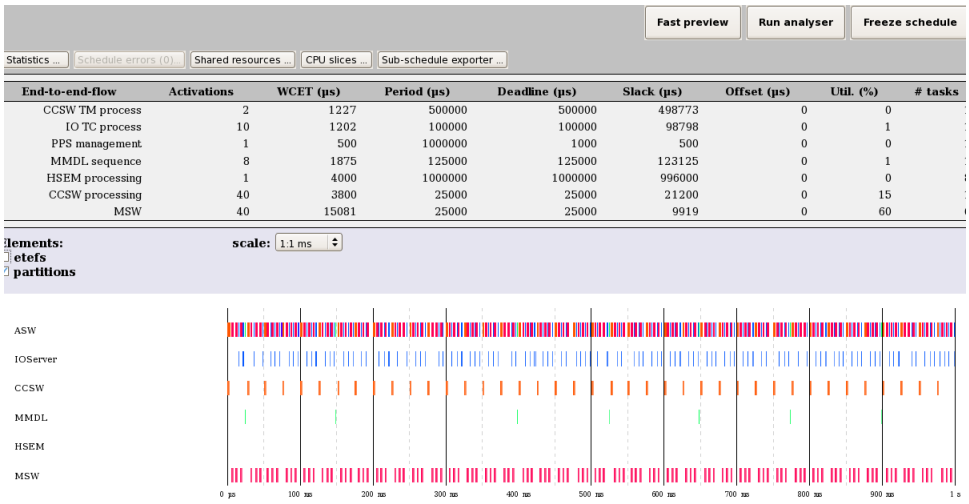

Figure 2-3: Result report of time slot to partition assignment

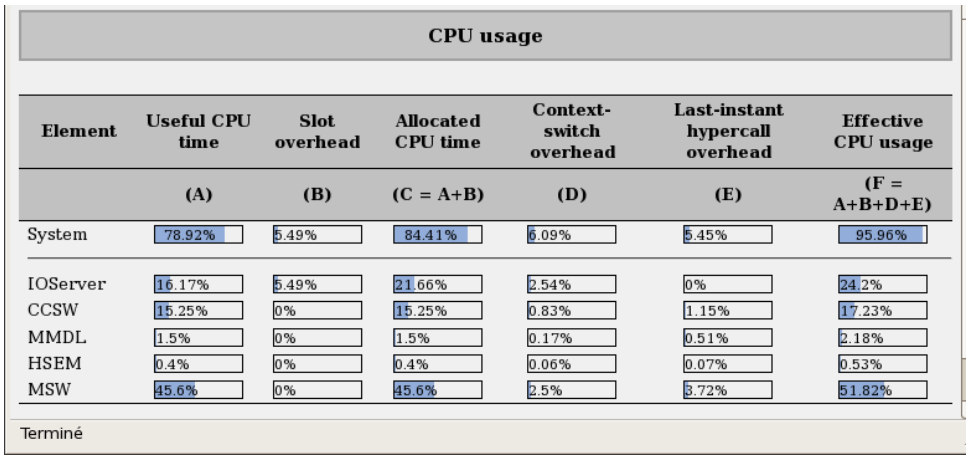

Figure 2-4: Partition CPU usage result report

2.3.3 Linux as XNG guest OS

The Linux kernel is modified to execute as guest OS in an XNG partition, thereby the applications, developed for Linux, can be executed. Having Linux inside a hypervisor partition provides the following advantages:

- Scalability and portability: Linux applications can be more easily ported to other hardware platforms supported by XNG.

- Robust features: The Linux kernel supports multithreading and has networking, graphics, and multimedia capabilities.
- Open source software: Huge amount of high-quality software available out-of-the-box with a supporting community.
- Device drivers: The device drivers abstract the system hardware to the user space applications. They can be built statically into the kernel or loaded dynamically at runtime as kernel modules. Embedded Linux distributions often contain I/O device drivers (GPIO, SPI, I2C, CAN, Ethernet MAC, USB, etc) directly provided by manufacturers.

2.4 Abstract communication concepts

2.4.1 Synchronous communication (request/response)

Synchronous communication according to the request-response approach is one of the basic methods that systems use to communicate with each other in a network. The first system sends a request for some data and the second one responds to the request. More specifically, it is a message exchange pattern in which a requestor sends a request message to a replier system, which receives and processes the request, ultimately returning a message in response. This is a simple but powerful messaging pattern, which allows two applications to have a two-way interaction over a channel; it is especially common in client–server architectures.

Accordingly, web service calls over HTTP hold a connection open until the response is delivered or the timeout period expires. HTTP functions as a request–response protocol in the client–server computing model. A web browser, for example, may be the client and an application running on a remote system hosting a web application may be the server. The client submits an HTTP request message to the server. The server, which provides resources such as HTML files and other content, or performs other functions on behalf of the client, returns a response message to the client. The response contains completion status information about the request and may also contain requested content in its message body.

However, request-response may also be implemented asynchronously, with a response being returned at some unknown, later time. When a synchronous system communicates with an asynchronous system, it is referred to as "sync over async" or "sync/async". This is common in enterprise application integration (EAI) implementations where slow aggregations, time-intensive functions, or human workflow must be performed before a response can be constructed and delivered.

The most dominant technology that is based on a subset of the HTTP is the REST software architectural style. It is commonly used to create interactive applications that use Web services, the so-called RESTful services. Such a Web service must provide its Web resources in a textual representation and allow them to be read and modified with a stateless protocol and a predefined set of operations. This approach allows interoperability between the software systems on a network that provide these services.

2.4.2 Asynchronous communication (event-driven paradigm)

The event-driven paradigm offers an alternative approach at control, communication, and optimization. The key idea is that an action is not dictated when a time is reached but when an event occurs. Such events are well-defined conditions of the system state. Communication-wise, an event occurs when a packet is transmitted and received at various distributed nodes.

Nowadays, many systems are networked and spatially distributed. In such settings, especially when energy-constrained wireless devices are involved, frequent communication among system components can be inefficient, unnecessary, and sometimes not feasible. Thus, rather than imposing a time-driven communication mechanism, it is reasonable to define specific events instead, which dictate when a particular node in a network needs to exchange information with another node. In stochastic environments, significant changes in the operation of a system are the result of random event occurrences, so that understanding the implication of such events and reacting to them is crucial. It is also interesting to note that event-driven approaches for sampling, estimation, and control tasks possess important properties. These properties are related to variance reduction and robustness of control policies to model uncertainties, which render them attractive compared to time-driven alternatives.

Following the principles of the event-driven paradigm, many integration patterns have evolved during the last decade for the integration of heterogeneous systems. The most dominant categories in this domain, however, are publish/subscribe architectures and event streaming.

In software architecture, publish–subscribe (pub/sub) is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers. Instead, published messages are categorized into classes without knowledge of which subscribers, if any, there are. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, exist. Publish–subscribe is a sibling of the message-queue paradigm, and is typically one part of a larger message-oriented middleware system. In that sense, the publish-subscribe model allows messages to be broadcasted to different parts of a system asynchronously. Unlike message queues, which batch messages until they are retrieved, message topics transfer messages with no or very little queuing, and push them out immediately to all subscribers. Compared to message queues, where message are received from specific subscribers in a round-robin fashion, in message topics the messages are received from all the subscribers unless message filtering policies applied by the subscriber itself. Most messaging systems support both the pub/sub and message queue models in their API. This provides greater network scalability and allows for a dynamic network topology. However, it decreases the flexibility to modify the publisher and the structure of the published data.

Indicative messaging middlewares that follow the publish/subscribe pattern that will be investigated include:

- ActiveMQ: Apache ActiveMQ is an open source message broker written in Java together with a full Java Message Service (JMS) client. It provides "Enterprise Features" fostering the communication from more than one client or server.

- MQTT: The Message Queue Telemetry Transport (MQTT) is an ISO protocol for messaging between IoT devices. Mosquitto and EMQX are two well-known open source brokers.
- RabbitMQ: RabbitMQ is a message queue. It doesn't strictly follow the pub/sub pattern. However, it can be configured for a direct or a fan-out message exchange between two or more components of the system.

Similar to publish-subscribe, event streams help software to extend beyond the request-response approach that is common to Web API styles such as REST, GraphQL, and gRPC. Software can communicate bi-directionally, removing the need for API consumers to continually poll for state changes. Instead, APIs publish events to an event stream for notification of data changes or important business events to any number of subscribed services for further processing.

The dominant technology in this domain is Apache Kafka, a distributed data platform that delivers higher throughput than traditional message brokers, such as ActiveMQ or RabbitMQ. Kafka removes the need for transactional messaging as found in message brokers, opting instead for turning events into message streams. These streams are accessible by any authorized subscriber and may be accessed in real-time or processed sequentially from a predetermined location or at the start of the stream.

2.4.3 API open specifications

Application Programmable Interfaces (APIs) define the way of interaction between software components. APIs define the calls or requests that can be made as well as the data types and formats that must be exchanged. An API simplifies programming by abstracting the underlying implementation. It only exposes objects or actions the developer needs.

The world of APIs is often one of competing standards, interests, and solutions. OpenAPI and AsyncAPI are two popular solutions that generate machine-readable documentation but for different types of APIs.

The OpenAPI Specification, formerly known as Swagger, is a solution that produces machine-readable documentation for REST APIs. As Swagger was developed and expanded, the OpenAPI Initiative was launched to further develop and promote the Swagger toolset in an open format, supported by major industry players to ensure standardization and support. In 2016, Swagger was officially renamed to OpenAPI Specification.

OpenAPI is language-agnostic and is used to automatically generate documentation for a wide set of functions, methods, parameters, models, and more. The main selling point behind OpenAPI is that unifying and standardizing the documentation in this way, the client libraries, source code, executables, and documentation are all natively synced, as the specification documentation itself is used to generate most of the rest. Through the declarative resource utilization, the end-user does not need to have any knowledge of server implementation, resources, access to server code, etc. to understand and consume the services.

AsyncAPI was initiated as to address the lack of tooling in the message-driven domain. Based on this need and the foundation of OpenAPI, focus of AsyncAPI is a solution to standardize message-based specification systems. The group behind it formulated a "theory of APIs" so

that AsyncAPI would serve as a common unifying language for many existing formats, protocols, and specifications, allowing for standardized communication in the message-driven field.

3 Software component programming model

As introduced in section 1, software component implementations developed according to the XANDAR process are to be processed by the XANDAR toolchain to be deployed both in a target-independent simulation for functional verification as well as on the target platform. To facilitate this, the system developer is restricted concerning usable programming languages and features to ensure compatibility with XANDAR concepts as well as the XANDAR toolchain. These restrictions are defined by the SWC programming model and form the basis for a SWC API.

3.1 Application structure

In XANDAR, the term application is used to refer to software which realizes a high-level functionality in the system under development, such as controlling an airbag or realizing a ground proximity warning in an aircraft. An application typically comprises interaction with sensors to obtain information on the embedded system's environment, data processing to determine the appropriate response action, and actuators to carry it out, i.e. influencing the environment.

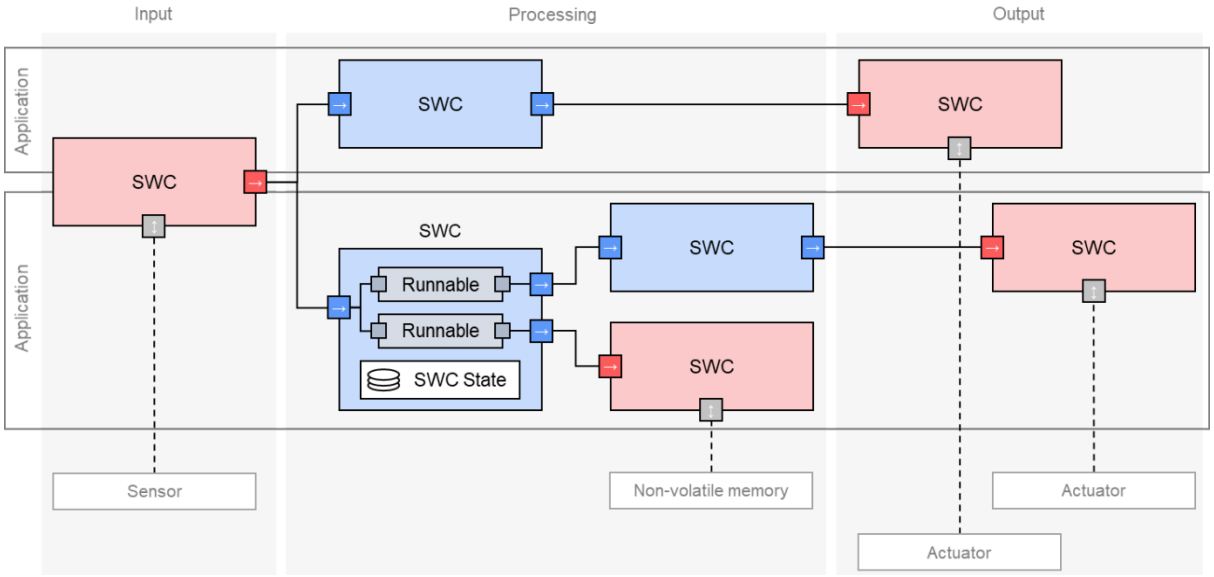

Figure 3-1: Exemplary realization of two applications in a software architecture

In the XANDAR development process, applications are realized as interconnected software components, each implementing a subset of the application functionality. Among these software components, there are input SWCs interacting with sensors, output SWCs interacting with actuators, and processing SWCs without environment interaction. Based on the classification introduced in section 1, input and output SWCs can be considered low-level interfacing components (LIC). Processing SWCs can be both LIC or processing components (PC), depending on their functionality. If a processing SWC only uses the CPU (and associated memory), it is considered a PC. In case that specialized hardware devices such as GPUs, AI components, or non-volatile memory are used, the SWC is a LIC. In that case, the SWC on the CPU controls the specialized hardware device.

Software components within an application are linked by unidirectional communication channels to enable information exchange, thereby forming a directed, weakly connected graph. It is possible that software components are shared among applications, for example an SWC providing sensor data for different functions. An exemplary case is depicted in Figure 3-1.

Each software component comprises one or multiple *runnables* and a SWC state which is shared among them. A runnable is an instruction sequence realizing parts of the SWC functionality. To obtain and transfer information, it interfaces SWC ports. Within a LIC, runnables can also interface specialized devices via a low-level device interface (LLDI). The condition on which a runnable is triggered for execution can be defined by the developer as described in section 3.2.2.

According to the XANDAR development process (cf. the excerpt in Figure 1-1), the system developer derives the software architecture from the logical architecture and the requirements. In this step, the software component structure and ports are defined as well as their runnables and runnable trigger conditions. During the generation of SWC skeletons, the XANDAR toolchain generates skeletons for each runnable. The system developer can then fill in the runnable code to implement the functionality. During system simulation and system generation, the XANDAR toolchain generates appropriate wrapper software, which integrates the runnables into the SWC according to the configured trigger behaviour, and then integrates the software components into the target environment, e.g. the simulation environment or the platform runtime execution environment.

In case of LIC, a transparent way to access low-level devices is needed, enabling access to the physical device on the target platform and integrating the appropriate device model during simulation. As mentioned above, access to low-level devices is realized by low-level device drivers (LLDD). These drivers provide a driver-specific interface to the system developer, which is referred to as low-level device interface (LLDI). The LLDI allows the system developer to access low-level device functionality during runnable development, e.g. by calling driver functions. For supported target platforms, the driver contains code which adheres to the LLDI and realizes the communication to the low-level device. This code may make use of services provided by the platform's base software, e.g. AUTOSAR. For simulation, the driver includes a Ptolemy II model describing the low-level device behaviour along with LLDI-compliant code interfacing the low-level device model via communication channels provided by the simulator.

3.2 Programming model specification

This section describes the model of the SWC interface, i.e. the general concepts that a developer must adhere to during the SWC code creation step as visualized in Figure 1-5.

3.2.1 Programming languages and language elements

As described in section 1, a fundamental aspect of the XANDAR design methodology is that every SWC implementation can be executed both in a model-based simulation framework (Ptolemy II) and on the target runtime environment (AUTOSAR, XtratuM, …). During the creation of SWC code, which will later be translated into a SWC implementation by the toolchain, the developer is therefore limited to the use of programming languages that are explicitly supported by the toolchain.

The most important property of a supported programming language is that suitable SIM and RTE adapters (as visualized in Figure 1-6) can be generated for each SWC that is specified as part of the relevant software architecture. More specifically, a SIM adapter that transforms the SWC implementation into an executable for the host computer executing Ptolemy II must be derived. Furthermore, the toolchain must generate an RTE adapter that transforms the SWC implementation into an executable for the target runtime environment.

The primary programming language that the XANDAR toolchain supports for the creation of SWC code is C. The integration of compiled programming languages that offer an interface to C and support the targeted runtime environment (such as C++ or Rust) are generally supported as well.

The SWC skeleton, which the toolchain will automatically derive from the software architecture, contains certain callbacks for every Runnable that is part of the SWC. These callbacks are available as C functions and need to be populated by the developer.

To populate the function bodies of all callbacks, the developer may use all language constructs that are supported by the targeted C standard with the following two exceptions:

1. Usage of standard library capabilities (string handling functions given by “string.h” header, memory management functions given by “stdlib.h”, …) must be limited to the support that the underlying target runtime environment provides.
2. The shared state of a SWC must be obtained and altered only via a data structure specified in C and managed by the respective SIM or RTE adapter. Most importantly, variables that according to the C standard [8] have a static storage duration must not be used from the callbacks or any function that is invoked from any of the callbacks.

In addition to the state management logic, the SIM and RTE adapters of a SWC provide a set of pre-defined C functions that must be used to interface with the input/output ports of SWC. For every low-level interfacing component, the adapters provide suitable C functions to access the devices that are attached to the low-level ports of the SWC.

3.2.2 Internal architecture of a SWC

This section builds upon the application structure as described in section 3.1 and defines the internal architecture of a SWC in more detail.

As mentioned above, every SWC comprises one or more runnables. In the XANDAR design methodology, a runnable is understood as a sequence of instructions. In the simulation, these instructions will be executed in the given sequence. From the perspective of the target runtime environment, the sequence may either be executed as it is or passed through specialized code transformation steps before the transformed sequence is executed on the actual target, which will be subject to further research in XANDAR. All the runnables that belong to a given SWC must be specified as part of the software architecture for the system under consideration.

Furthermore, a data structure is specified for every SWC that encapsulates all state variables that are relevant for the execution of its runnables. In contrast to runnables, this data structure is not part of the software architecture. Instead, it is an implementation detail that the developer is expected to introduce when creating the SWC code.

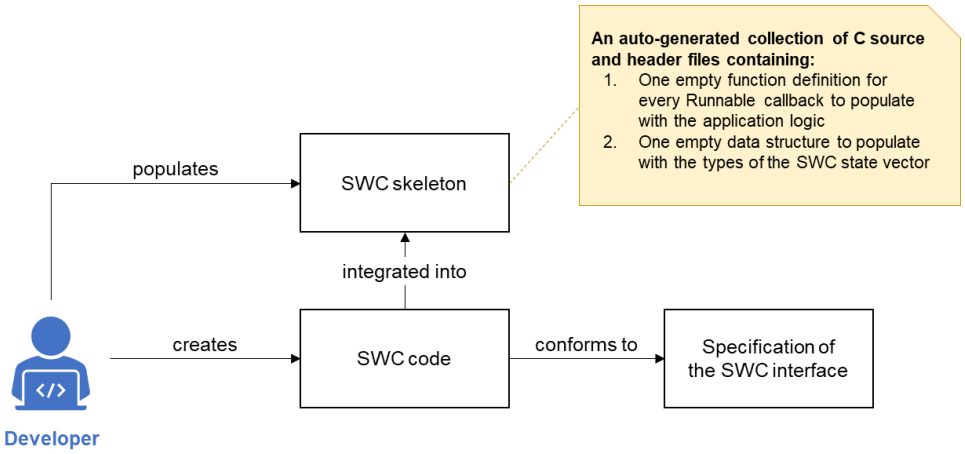

Figure 3-2: Nature of the automatically generated SWC skeleton

Figure 3-2 is an updated version of Figure 1-5 and visualizes the nature of the SWC skeleton that the toolchain automatically generates for every specified SWC in more detail. As shown in the figure, the SWC skeleton consists of a collection of C source and header files. In combination, these files provide the developer with an empty function declaration for every runnable callback as well as with one empty data structure that is supposed to capture the composition of the state vector of the SWC under consideration.

More specifically, two callbacks are provided per runnable:

1. **Initialization callback:** This function is triggered exactly once and only during the setup phase of the SWC. Most importantly, developers may use this callback to initialize the state variables that are relevant for this runnable.
2. **Trigger callback:** This function is called whenever the trigger condition of the runnable is met. As part of this callback, developers may obtain values from input ports of the SWC, provide values to output ports of the SWC, interact with low-level devices, and read from or manipulate the state vector.

Except for the state vector, which the SIM or RTE adapter will explicitly provide to the SWC, the trigger callback is expected to be stateless. Most importantly, this means that no variables with a static storage duration may be accessed from it.

Trigger conditions for runnables must be defined as part of the software architecture. As part of the XANDAR design methodology, two types of conditions are supported:

1. **Periodic triggers:** The respective runnable will be executed periodically and with a fixed duration between the repeated invocations.
2. **Data event triggers:** The respective runnable will be executed whenever data is available at one or more input ports of the SWC.

The functions to access input and output ports, which are provided by the respective adapters, guarantee a non-blocking operation, i.e. non-blocking reads from input ports and non-blocking writes to output ports. A detailed description of this can be found in section 3.2.5.

3.2.3 In-SWC parallelism

As described in section 1, the XANDAR SWC should run both on a model-based simulation framework (Ptolemy II) and on the target runtime environment (e.g., AUTOSAR, XtratuM). This makes annotation-based parallel programming languages, such as OpenMP, an excellent choice for the XANDAR SWC implementations. This is due to the fact that in annotation-based languages unsupported pragmas (e.g. no such library is supported, or only one CPU exists) are simply ignored by the compiler. Thus, the code can be executed perfectly fine, as if the pragma was not there.

The two most popular annotation-based parallel programming languages are OpenMP and OpenACC. Both support C/C++ and Fortran as well as CPUs and hardware accelerators. They consist of a set of compiler '#pragmas' that control how the program works. A '#pragma' is a compiler directive that tells the compiler to do something special beyond the scope of standard C/C++/Fortran language. Directive specifics are '#pragma parallel', '#pragma for' etc. The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behaviour, but without any parallelism.

Comparing to the OpenACC, the OpenMP directives instruct the compiler to optimize the code in a more specific way (the programmer can provide more low-level details), leaving little to the discretion of the compiler and its optimizer. This leaves more freedom for the developer through better interaction with the hardware and the ability to optimize code. In turn, parallelization and optimization of the code are in the programmer's responsibility and the runtime cannot verify that the generated code is correct. It is also the programmer's responsibility to avoid race conditions or to make sure that the code generates the expected output. On the other hand, OpenACC supports less directives and leaves the optimization to the compiler.

OpenMP and OpenACC target both CPUs and hardware accelerators. In the first case, they can generate multi-threaded code and vectorized code, leveraging every aspect of the CPU's parallelism. OpenMP 5.0 also supports code transformation directives (for example loop tiling or loop unrolling), which ease and enhance the optimization process.

It is obvious that the aforementioned parallel programming languages represent an excellent choice for the XANDAR toolchain and our proposal is to make use of one of the two standards. Before the final choice, however, a more detailed investigation of the named options is planned.

Other non-annotation based parallel languages can be supported too, such as CUDA and OpenCL. It is important to note that such languages cannot run on model-based simulation frameworks such as Ptolemy II and thus, they can be used only when a C/C++ implementation is accompanied. For example, two implementations can be provided, a CUDA and a C-code version of a function (either serial version or annotated parallel version). The usage of OpenCL will be investigated if the target platform supports specialized hardware accelerators.

3.2.4 GPU/AI unit integration

In XANDAR, the inference (runtime) part of the neuromorphic applications is handled as a regular SWC, thus it follows the skeleton and communication pattern as described in section

1 of this deliverable. The inference part of neuromorphic applications will be extracted from a high-level AI/ML framework as a standalone SWC in C/C++ programming language and will be fed to the XANDAR toolchain as a common software implementation with specified input and output ports. This way, even the neuromorphic SWC can be wrapped into a Ptolemy II actor, which allows to simulate its behaviour.

However, from the design methodology perspective, there is one main difference between neuromorphic applications and other SWCs. That is, each neuromorphic SWC provides different functional properties. While the main topology of all neuromorphic applications is kept, we will provide different implementations per neuromorphic SWC by varying its data type. This means that each neuromorphic SWC is provided in at least two ways with varying data types, e.g. FP32 and FP16 or INT32 and INT8. In consequence, accuracy and prediction vary in dependence of the chosen variant. Which implementation matches best is decided by the functional analysis and verification phases of the XANDAR toolchain.

Concerning the higher-level AI/ML framework, the only requirement that it is posed by our approach is a C/C++ implementation of the inference part of a neuromorphic application. This is required by the modelling and verification parts of the project. In the proposal TVM/NNVM is listed as a potential option that can be adapted by the XANDAR project (due to its strong machine independent and machine dependant model transformation and optimization capabilities). After careful examination of the TVM/NNVM framework, we realized, however, that it is not possible to extract a C/C++ implementation of a neuromorphic application using this framework. We also investigated two other widely-used AI/ML frameworks (TensorFlow and PyTorch) with the following result. TensorFlow includes stable C/C++ generators for both 64-bit and 32-bit architectures. The corresponding feature in PyTorch is still unstable and not mature. However, it should be noted that this does not exclude the usage of PyTorch in the XANDAR project as a framework for training of an AI/ML model. This is because there are various industrial formats (e.g., the ONNX format) that allows to port a trained model from PyTorch to TensorFlow which allows to use the TensorFlow C/C++ generators.

Finally, it is important to mention that in case the final implementation (for the final platform) uses a specific AI or GPU hardware accelerator, we are able to use TensorFlow or PyTorch for the extraction of the inference code for a particular SW API, e.g. CUDA or OpenCL API. The actual dispatching of a neuromorphic application to an AI or GPU hardware accelerator is subject to decision in a later phase of the project. Currently, we have tested all three abovementioned frameworks (TVM/NNVM, TensorFlow, and PyTorch) and it is possible to extract the final inference code in all well-known APIs.

3.2.5 Inter-SWC communication

Communication between software components is realized via directed, message-based communication channels as provided by the base software of the respective platform or the simulator. In XANDAR, two types of communication are supported, implemented by queuing ports and sampling ports.

Communication via queuing ports adheres to the FIFO (first-in first-out) concept. The order of messages is preserved and messages are buffered until consumed by the receiver. The XANDAR programming model guarantees that sending a message via a queuing port is a non-

blocking operation. Therefore, it does not impose limitations on the number of messages that are queued for delivery in a channel. However, since such limitations are naturally present in real-world systems, this buffer capacity needs to be considered during system generation. Reading from a queuing port is also guaranteed to be a non-blocking operation. It always consumes a message from the queue. Reading messages without consuming them is not supported. If the FIFO is empty, this situation is indicated and no element is read. With respect to software execution control, runnables can be configured to run when a message is available on an input queuing port.

Communication via sampling port follows the last-is-best principle. If multiple messages are written to a channel before a read, the latest message is read. Earlier messages are discarded. Multiple read operations return the same message until another write has occurred. If no message has been written to a communication channel since initialization, the read operation indicates that no data is available. Thereby, the programming model guarantees that both write and read operations are non-blocking.

As introduced in section 1, the software architecture specifies all SWCs which are present in the system as well as their input and output ports, port types, and the communication channels between them. Based on this configuration, the XANDAR toolchain generates the appropriate intermediate code to realize the described communication behaviour for both the simulation and the target platform.

3.2.6 Safety and security considerations

This section outlines the importance and needs for incorporating safety and security artefacts within the SWCs to harden the networked embedded system defences.

3.2.6.1 The need for security and safety artefacts

The mixing of critical and non-critical software components brings significant advantages within the networked embedded system such as consolidation of different functionalities within a single system, reducing costs, and increasing efficiency. But at the same time, mixed-critical software brings major security and safety challenges due to sharing of critical system control and data. The advanced embedded multiprocessor system-on-chip architectures allow implementation of mixed-critical software applications and provide on-chip security mechanisms to segregate and protect system resources, right from the powering on of the system, to prevent misuse and compromise. However, these security and safety measures have been found vulnerable due to a lack of secure software design practices and the adoption of ad-hoc or passive defences.

To approach this, an X-by-Construction approach is adopted in XANDAR. The idea of X-by-Construction is to bake/incorporate safety and security defences rather than an afterthought and deploy them across various layers of the overall software system of the networked embedded system. These artefacts would facilitate building the SWC-level trust boundaries to realize use-case specific safety and security model, as well as monitoring and control runtime activities.

3.2.6.2 Purpose of security and safety artefacts and their interaction with the system

The objective of the safety and security artefacts is to establish a means to regulate/monitor SWC activities at its entry/exit points closer to the source as illustrated in Figure 3-3. All the control and data traffic transmitted to and received by the SWC and its interaction with other system software layers shall pass through this controlled and monitored interface.

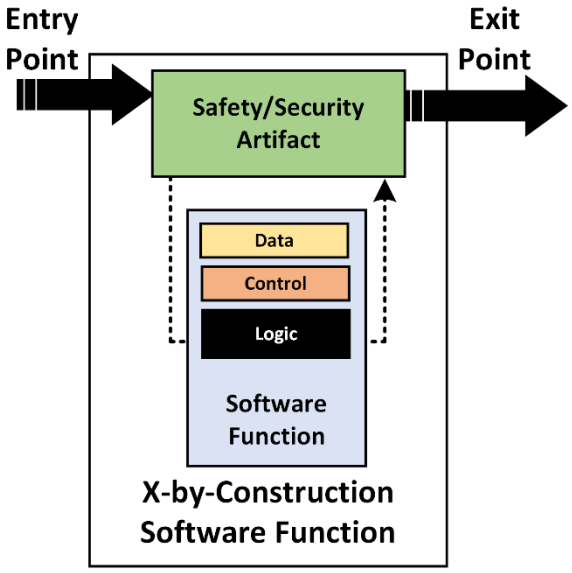

Figure 3-3: Safety/security monitor interaction of a software component

Based on the security engineering principles, the management and control of safety and security artefacts shall be handled by an independent system-level entity (preferably the hypervisor) and shall be transparent to the SWC developer to the extent possible. Furthermore, the realization of safety and security artefact can be either in hardware or software, which is subject to further research in XANDAR.

4 Summary

In the development of embedded systems according to the XANDAR approach, the programming model lays the foundation to enable both the verification of functional requirements by simulation and the realization of non-functional requirements such as safety and security by pattern-based model and code transformations. This is achieved by a design methodology, where application code templates are generated from software architecture models, populated by the developer, and then integrated both in the simulator and the actual hardware platform. The integration of the application code into the respective target is realized by generating adapter code for simulation and the respective RTE of the embedded platform.

To enable this, the programming model structures applications in software components and runnables, which communicate by communication ports and channels. To access low-level devices, the concept of low-level interfacing components is introduced. These concepts can be realized both during simulation and on the embedded target platforms. They especially allow code and model transformation concepts to fulfil safety and security requirements. As target RTEs, AUTOSAR and the XtratuM hypervisor are envisioned, which can be used to host the application software and realize the necessary inter-SWC communication.

5 References

[1] AUTOSAR initiative, „AUTOSAR Website - Standards“, “AUTOSAR Methodology,” November 2020. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/20-11/AUTOSAR_TR_Methodology.pdf. [Accessed 26 May 2021].

[2] AUTOSAR initiative, „AUTOSAR Website - Standards“, “Methodology for Adaptive Platform,” November 2020. [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/adaptive/20-11/AUTOSAR_TR_AdaptiveMethodology.pdf. [Accessed 17 June 2021].

[3] AUTOSAR initiative, „AUTOSAR Website - Standards“, “Specification of ECU Configuration,” [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/20-11/AUTOSAR_TPS_ECUConfiguration.pdf. [Accessed 26 May 2021].

[4] AUTOSAR initiative, „AUTOSAR Website - Standards“, “Specification of RTE Software,” [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/20-11/AUTOSAR_SWS_RTE.pdf. [Accessed 26 May 2021].

[5] Vector Informatik GmbH, “MICROSAR,” [Online]. Available: https://www.vector.com/gb/en-gb/products/products-a-z/embedded-components/microsar/. [Accessed 26 May 2021].

[6] Elektrobit Automotive GmbH, “EB tresos product-line,” [Online]. Available: https://www.elektrobit.com/products/ecu/eb-tresos/. [Accessed 26 May 2021].

[7] The MathWorks, Inc., “AUTOSAR Blockset - MATLAB & Simulink,” [Online]. Available: https://uk.mathworks.com/products/autosar.html. [Accessed 26 May 2021].

[8] International Organization for Standardization, *ISO/IEC 9899:2018: Information technology — Programming languages — C,* 2018.