Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Dependability assurance in cyber-physical systems

Scientific Students' Association Report

Author:

András Földvári

Advisor:

Prof. Dr. András Pataricza (BME)
Dr. Josef Pichler (SCCH)

2018

Contents

Kivonat

A kiber-fizikai rendszerek (Cyber-Physical Systems - CPS) a fizikai világ és az informatika összekapcsolódásaként jönnek létre. Növekvő fontosságuk megjelenik az élet több különböző területén, legyen az akár egészségügy, szállítmányozás vagy okos gyártás. A nem megfelelő működésük kritikus hibához vezethet, ezért fontos a megfelelő tervezésük és megvalósításuk. A dolgozat célja egy módszer létrehozása volt, amivel biztosíthatók az integritási, a biztonságossági és a megbízhatósági követelmények a tervezéstől egészen a működtetésig.

A CPS-k tervezése és üzemeltetése összetett feladat. A kritikus CPS megfelelő szolgáltatásbiztonságának eléréséhez elengedhetetlen a számos funkcionális és extra-funkcionális követelmény teljesítése.

A mérnöki világban bevett gyakorlat a különböző módszertanok használata a rendszer tervezése során. Ezek lehetnek ajánlások vagy egy adott szervezet által bevált gyakorlatok, mint a NASA System Engineering Handbook vagy az NIST CPS Framework. Hasonlóképpen, a de-facto és ipari szabványok által leírt különböző referencia-architektúrák és platformok támogatják az komponens alapú rendszerintegrációs szolgáltatások megvalósítását.

A kritikus rendszerek formálisan bizonyítható helyessége egyre fontosabb a gyakorlatban. Egy összetett CPS számos komponenst tartalmaz az architektúrális szinteken belül és azokon átívelően is. A komponensek közti kölcsönhatások lehetnek a fő hibaforrások a rendszer integrációja során, ezért a komponensek általános tesztelésén kívül elengedhetetlen az integráció ellenőrzése is. A dinamikus CPS-k szolgáltatásbiztonságának garantálása szükségessé teszi az interoperabilitás ellenőrzését a tervezéstől a működtetésig.

A dolgozat célja a komponens integráció tesztelése, valamint a futás idejű verifikiáció támogatása volt, a komponenstesztek felhasználásával. A módszer kiegészíti, az elsőként a NASA által a 90-es években használt Assume-Guarantee tesztelési elvet egy általános célú futás idejű paradigmával.

A módszer támogatja a rendszer integrációhelyességének ellenőrzését, felhasználva a komponensek formális leírását (temporális logikai kifejezések, állapotgépek) és a rendszerre vonatkozó működési feltételezéseket a komponenstesztekből kiindulva.

Az integráció ellenőrzése az egyes komponensek tesztszekvenciáinak összefűzésével, valamint a működési feltételekre vonatkozó feltevések (előfeltételek) és teszt kimenetelek (utófeltételek) ellenőrzésével végezhető el. Az állapotgépeken alapuló egységes modell-alapú megközelítés végrehajtja a szekvenciák, az elő- és utófeltételek ellenőrzését a komponensekre és segíti a kódgenerálást a futás idejű ellenőrzéshez.

A Data Distribution Service (DDS), az Industrial Internet Consortium által is ajánlott, kiber-fizikai rendszerekben előszeretettel használt OMG szabvány szolgált a dolgozatban bemutatott tesztrendszer alapjául.

Abstract

Cyber-Physical Systems (CPS) are smart systems integrating the physical and the computational world of increasing importance in a variety of domains, like healthcare, transportation, smart manufacturing. As their malfunctions can lead to critical incidents, their proper design and implementation are essential. My research objective was the elaboration of a method assuring the integrity, safety and reliability requirements of the system at design and run-time.

Building and managing CPSs is a very complex task. Dependability of a service delivered by a critical CPS necessitates the fulfillment of a variety of functional and extra-functional requirements.

System engineering has a variety of well-established design methodologies corresponding to the best industrial practice like NASA System Engineering Handbook or the NIST CPS Framework. Similarly, different reference architectures and platforms described by de-facto and industrial standards support component and (sub)system integration-based service implementation.

Formal proof of correctness of critical systems is a growing practical importance part. A complex CPS can include a multitude of components within and across architectural levels. The interaction between components is the primary source of faults in integration-based systems composition. Integration testing is mandatory, even if the components are typically pretested. Dependability assurance in dynamic CPS necessitates the extension of checking the interoperability from design-time to run-time.

My objective was to support the component integration by reusing component tests for integration testing and run-time verification. Our method extends the Assume-Guarantee approach for testing elaborated initially by NASA in the 90's to a general-purpose run-time verification paradigm.

The new method supports the checking of the correctness of system integration starting of the formal description of the components (temporal logic expressions, state machines), the assumption on the operating environment by reusing the specific component tests.

Integration testing is carried out by waiving the test sequences of the individual components and simultaneously performing a formal check of the fulfillment of the assumption on operating conditions (pre-invariants) and the test oracles (post-invariants). A uniform model-based approach based on statecharts executes all the sequence, operating condition and output checking of the different components and facilitates the automated code generation of run-time checkers.

The OMG standard Data Distribution Service (DDS), a favorite candidate architecture for complex CPSs promoted by the Industrial Internet Consortium served as a testbed of the pilot prototype.

Chapter 1

Introduction

1.1 Objective

Dependability of a service delivered by a critical cyber-physical system (CPS) necessitates the fulfillment of a variety of functional and extra-functional requirements. The research objective was the elaboration of a method assuring the dependability of the designated system at design and runtime.

Complex CPSs consist of several third-party or reused components. Integrating the different services into the system is a challenging task. Usually, only the interface definition and specification (input-output invariants) is available. Documentation or detailed description lack on the internal functioning and implementation for third-party commercial of the shelf components. Accordingly, they operate as black-box components. For the integration of these components, the information about the services and optionally about the pre-defined test cases are available.

However, acceptance and compliance tests of the component are typically provided to or elaborated by the system integrator. This provides a valuable asset for assuring dependability of the system by supporting system integration. Moreover, as the report will illustrate it, they can be reused as runtime verification in addition to integration testing.

The starting point of my method is a well-founded approach for system integration developed originally by NASA which forms the foundation of their technology[29]. The existence of good test assuring a proper coverage is a prerequisite of integration based synthesis of critical applications. The idea was reusing them as the basis of runtime verification by embedding them the target system.

A fundamental method for exploiting the hierarchy of problems in modeling, testing, and verification, the assume-guarantee approach was extended to a general-purpose runtime verification paradigm in data-flow oriented cyber-physical system design. The method also supports the checking of the correctness of system integration starting of the formal description of the components (temporal logic expressions, state machines).

The engineering part of my work was to develop a model testing and runtime verification framework over the DDS middleware standard. The presented static and dynamic verification method use statecharts rely on the framework.

The first chapter presents a quick overview of cyber-physical systems (CPS) and a case study that is the running example in this report.

1.2 Context: Cyber-Physical Systems

Cyber-physical systems (CPS) are smart systems. They consist of physical and computational components interacting through communication layers: Transducers, i.e., sensors and actuators form the interface to the physical word.

Frequently, intelligent components (e.g., FPGAs, microcontrollers with network interfacing capabilities) or even complete embedded systems establish the connection between the physical world and cyberspace.

The computational components execute the business logic and the algorithms for data processing. The extensity and sophistication of services in the CPS world can be further increased by interconnecting individual CPSs to a larger one (System of Systems - SoS).

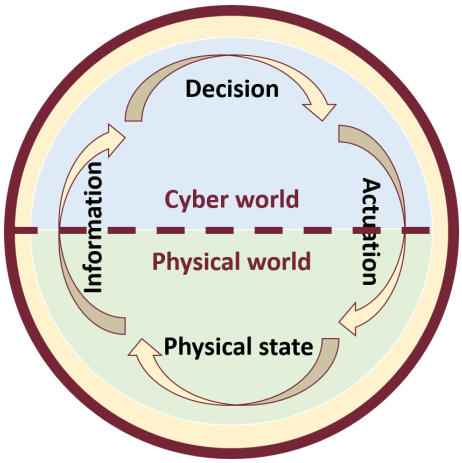

Figure 1.1: Cyber-Physical System

The potential use cases of CPS cover very different domains from manufacturing through autonomous cars to the smart ecosystem. Many domains invest in CPS to use and develop the technology. The global demand for reliable, interoperable and secure systems is rising.

To design and manage CPS is a complex task. The actuators influence the physical word depending on the information from the previous data (control loop behavior). Errors in the computational word are amplified by their potential physical impact and could be lead to damages in the physical word.

CPS design is a continually evolving area, so there is no consensus about a complete terminology. The report adopts the terminology in [34].

There is an increasing need for creating and operating CPSs. Ensuring the dependability of the critical services involves fulfilling both the functional and extra-functional requirements at the same time.

The conceptual design of CPS (Figure 1.2) shows that there is a continuous interaction between systems, components, and users. Several domains need a permanent connection to supervise or interact with the system.

CPSs consists of

- **sensors**: devices that monitor specific aspects (e.g., temperature, pressure) of the physical world and represents it digitally);

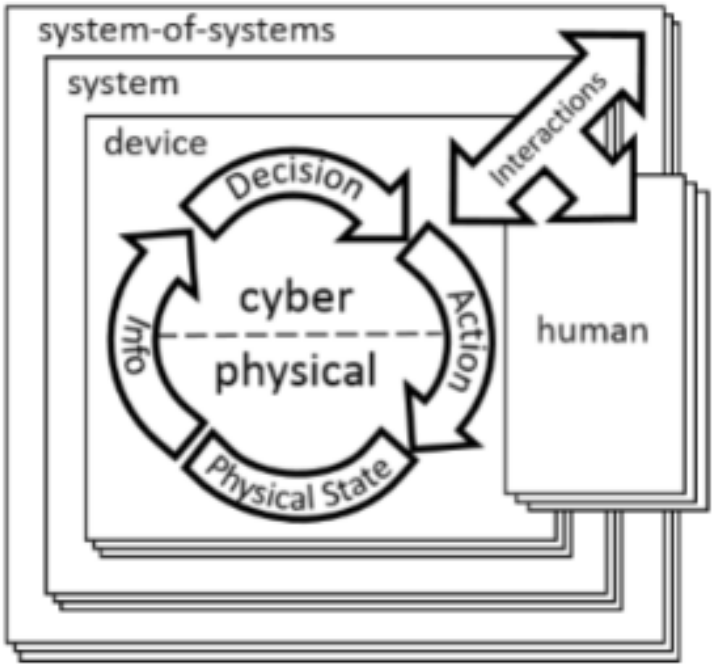

Figure 1.2: Conceptual Design of Cyber-Physical Systems[26]

- **actuators**: devices that can change the physical property of an element according to the inputs (e.g., heating);
- **communication** and **logic components**: provide time-critical functionality and high computational power.

The next section presents a CPS pilot example that was used in the report to demonstrate the goal of the mission statement.

1.2.1 Case study: Crossroad Traffic Lights

The case study is about a crossroad traffic light controller rely on the tutorial for Gamma Statechart Composition Framework [42] [5]. The model shows the crossing of a priority and a secondary road (Figure: 1.3). In each direction, the traffic lights are the standard 3-phase lights looping through the green-yellow-red-green sequence. Additionally, the model has a police interrupt mode. In this mode, the traffic lights blinking in yellow, and it is not possible to change the colors without another police signal.

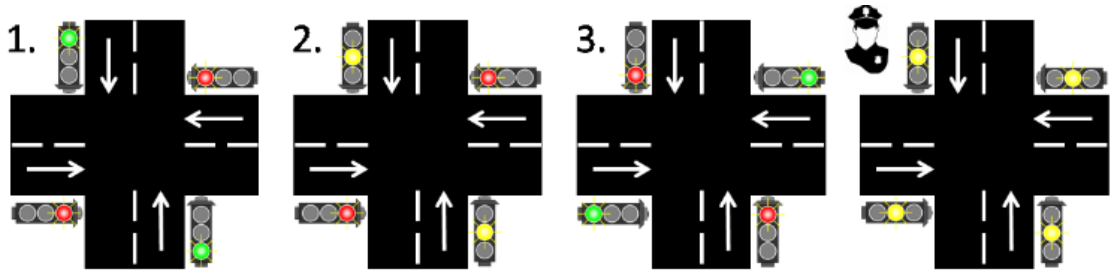

Figure 1.3: Crossroad

The goal was to present the integration of the components and show if any component violates the requirements. Section 8.6 presents how the runtime verification works on this example.

The interfaces of the components are well defined. The controller unit toggles the states of each traffic lights and an external actor can toggle the police interrupt signal.

This example is suitable for presenting the assume-guarantee runtime verification method because it is easy to create an error situation (e.g., both the crossroad lights are green or malfunctioning police interrupt) to show the opportunities of this approach.

Furthermore, the crossroad has and additional third-party camera component with classification service. The camera unit recognizes the different type of cars using neural network. This part of the example presents the integration of third-party components and shows how this approach can provide fail-safe or fail-silent behaviors.

1.3 Structure of the Report

However, CPSs are special systems, general design and quality rules can apply to them. Chapter 2, 3, 4 present the general and special parts of complex cyber-physical systems using requirement-based design approach. CPS design approaches and mathematical formalisms are also introduced here.

Chapter 5 describes a static verification method. This method provides algorithmic support for the design-space exploration helping the designer to maintain a clear view of the alternate architectural candidates.

Chapter 6, 7, 8 first describe the assume-guarantee approach to give an overview of the basis of the design-time and runtime verification methods. The *Design-Time Verification* chapter describe a method to present the verification and testing in system integration. The *Runtime Verification* chapter presents the assume-guarantee runtime verification method and describes a pilot implementation and example over DDS.

Chapter 9 summarize the report and presents possible further research.

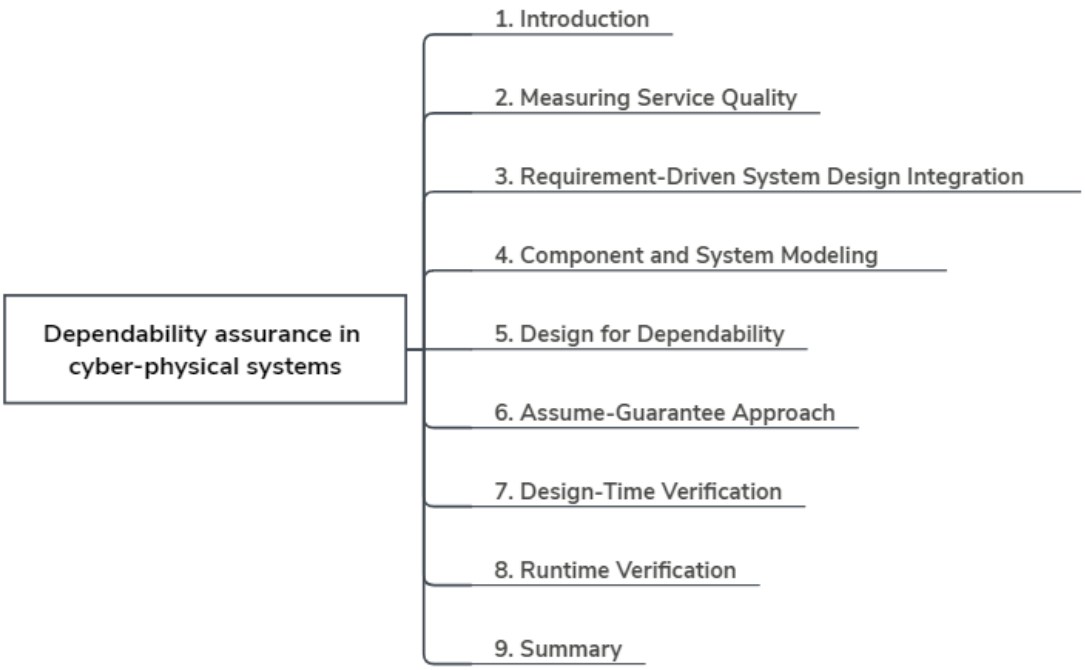

Figure 1.4: Contents

Chapter 2

Design for Service Quality

This chapter presents the quality characteristics of software-based systems and the main impact of dependability attributes (e.g., reliability, availability) onto it. Several dependability aspects have a high impact on software system quality.

This chapter also shows the importance of the functional and extra-functional requirements and the differences between them. Requirements define the structure of the system and help to define components with gradual refining the requirements. Selecting the proper requirements is crucial during the design and the implementation of the system.

2.1 Product Quality

2.1.1 Evaluation of Software Product Quality

The goal of the ISO/IEC 25000 (SQuaRE – System and Software Quality Requirements and Evaluation) standard is a framework for the evaluation of product quality.

The standard is the result of the evolution of several other standards (quality model for software product evaluation, the process for software product evaluation). The standards of ISO/IEC 25000 consists of five divisions. The most relevant standards to the current report are those by the Quality Model Division (ISO/IEC 2501n), especially the ISO/IEC 25010 [31] standard for System and software quality models.

The ISO/IEC 25010 quality model is the cornerstone of a product quality evaluation system. The quality model determines the set of quality characteristics taken into account when evaluating a software product.

> *The **quality of a system** is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value. Those stakeholders' needs (functionality, performance, security, maintainability, and so forth) are precisely what is represented in the quality model, which categorizes the product quality into characteristics and sub-characteristics and typically assigns metrics and acceptance criteria.*

The product quality model defined in ISO/IEC 25010 comprises the eight quality characteristics shown in Figure 2.1.

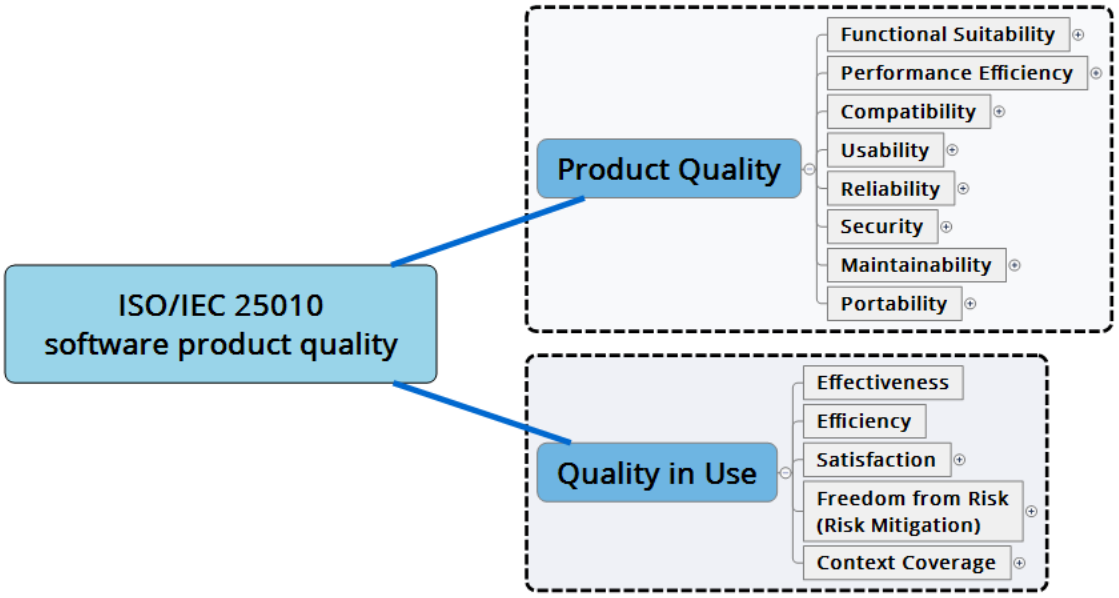

Figure 2.1: ISO/IEC 25010

2.2 Design Workflow

Usually, traditional design methodologies (Figure 2.2) need an extension to the CPS specific requirements as detailed later.

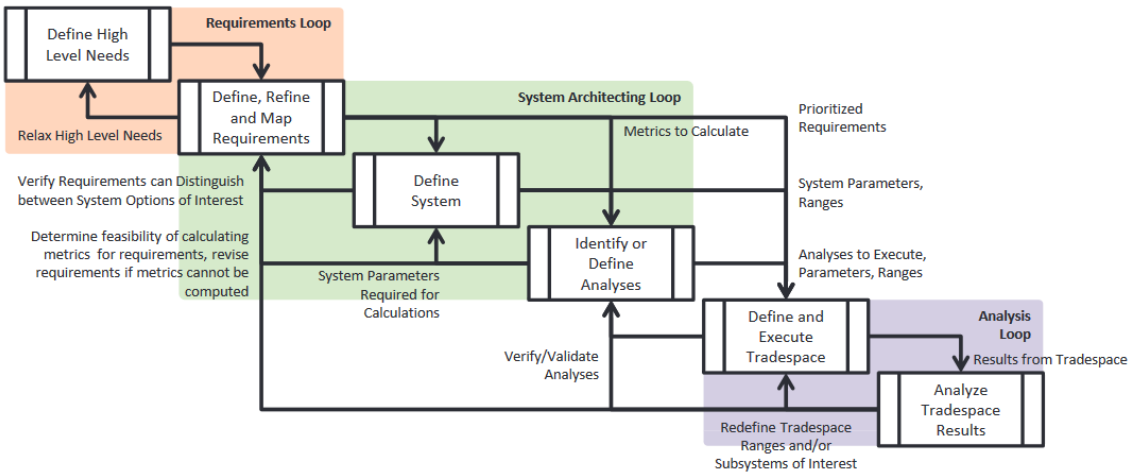

Figure 2.2: System Engineering Workflow[14]

CPS design workflow recommendations (e.g., Figure 2.3) are incomplete because it is an emerging domain with rapid changes. There is still ongoing research about the topic.

Model-based engineering paradigms are widely spread across CPS-targeted tools and methods. For instance, the objective of the *Integrated Tool Chain for Model-based Design of Cyber-Physical Systems* (INTO-CPS) [35] initiative the followings: tool chain for model-based design of CPS supporting the full development life cycle and guidelines and patterns to support the tool chain.

This workflow uses SysML[43] as a primary modeling language. SysML is a general-purpose modeling language for system engineering. SysML is capable of modeling physical

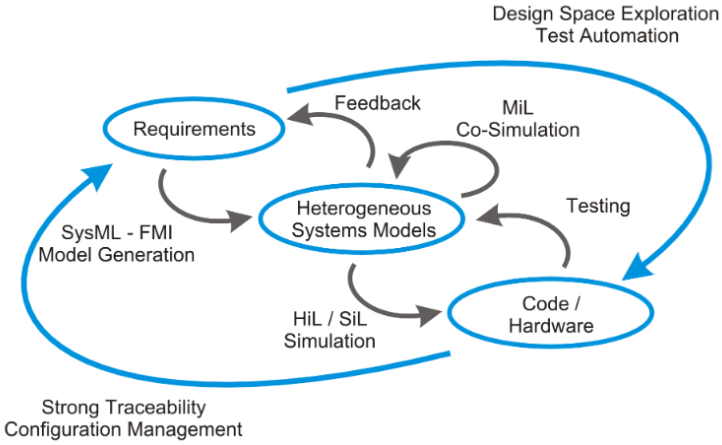

Figure 2.3: CPS Workflow[35]

(environmental) aspects (e.g., physical aspects like mass, pressure, and so forth). SysML is suitable to model complex CPSs.

The HIL (Hardware-in-the-Loop), SIL (Software-in-the-Loop) and MIL (Model-in-the-Loop) Co-simulation integration techniques are well fit into the workflow of this report.

Traceability shows the relation through the whole system development artifacts. It interconnects the requirements to the model instances and the implementation artifacts. Trace links can be used for verification purposes too.

This report reuses this requirement-based design approach focusing on the verification and validation part of the design workflow.

2.2.1 System of Systems

A system provides a specific service for a group of end users. For instance, a weather forecast system delivers its outputs to a large population of users. However, it can be integrated as a component into an agricultural or building facility management system. Generally, the system is a collection of components that work together to achieve a complex service (functionality, behavior) that the components themselves cannot exhibit alone. Each component of the system can be an individual system. This means System of Systems (SoS) describe a hierarchical composition of systems and components. The SoS is integration of the individual components.

Maier [38] provides the following definition:

> *A SoS is an assemblage of components which individually may be regarded as systems, and which possess two additional properties:*
>
> - ***Operational Independence of the Components:*** *If the system-of-systems is disassembled into its component systems the component systems must be able to usefully operate independently. That is, the components fulfill customer-operator purposes on their own*
> - ***Managerial Independence of the Components:*** *The component systems not only can operate independently, they do operate independently. The component systems are separately acquired and integrated but*

maintain a continuing operational existence independent of the system-of-systems.

The methods in this paper are capable to perform SoS integration over heterogeneous systems. However, the systems are heterogeneous, the general principles originating in the CPS framework middlewares (see Section 3.3) are similar.

2.3 Functional and Extra-Functional Requirements

As requirements formed the core elements in a CPS design workflow (Figure 2.3). ISO 26702 standard [32] defines the requirement as follows:

> *A requirement is a statement that identifies a product or processes operational, functional, or design characteristic or constraint, which is unambiguous, testable, or measurable and necessary for a product or process acceptability.*

Functional requirements specify the functions of the system or its components. The functions describe the behaviors which the system as a whole should satisfy and use cases where the system should operate. The functions usually involve computing processes, data processing and other functional behaviors that the system should achieve.

Extra-functional (non-functional) requirements are as critical as functional requirements. They typically cover the availability, reliability, maintainability, safety, and security aspects. Furthermore, they specify criteria that decide the appropriateness the operation of a system. Usually, extra-functional requirements used to define the boundaries of the system. They define restrictions (e.g., the proper operating temperature of the system).

One of the earliest and important design phase is to define the functional and extra-functional requirements.

The correct and complete set of requirements is essential to fulfilling the needs of the stakeholders. The well-refined requirements help to determine the components responsible for satisfying each sub-requirement.

CPS Specific Requirements As mentioned, traditional design methodologies are insufficient for designing CPS due to the complex functionality envisaged. This complexity originates in different aspects [26]:

- The relation between the physical and computational components
- Interconnection of systems (System of Systems) [17]
- Quality of Service (QoS) requirements: description or measurement of the overall performance of a service influenced by different domains for computation and physical world
- Timeliness aspects of real-time operation (reaction to critical situations that could cause harm)

- Scalability, interoperability, integrity aspects (e.g., integrating third-party services)
- Fault-tolerance: to assure the avoidance of failures in a critical application by means of some redundancy (e.g., failure of critical actuator devices)
- Human interaction: defines how actors interact with the system (e.g., Human-Machine Interface)

Modeling the Requirements In these days, the most used requirement authoring tools are textual documents, but professional tools (IBM Doors[6], MagicDraw[8]) can provide extra features (e.g., hierarchical decomposition, traceability, reference binary files, relations between requirements) and support interoperability in tool chains. This way all tool supporting a specific design step can rely on a single collection of requirements.

For instance, the ReqIF (Requirement Interchange Format) OMG standard[10] that used to can connect various tools.

2.3.1 Static and Dynamic Requirements

Requirements separate into static and dynamic types.

Static requirements could describe both platform specific and application specific properties (e.g., throughput, reliability, integrity). The configuration of the platform-specific aspects influences the requirements, thus the dependability.

Dynamic requirements describe the behavior of the service. The gradual refinement of the dynamic requirement could be the source of errors. Use cases capture all the cases where the system should operate. This is one of the highest abstraction levels of the dynamic requirements.

Static and dynamic modeling methods are available to model and verify this kind of requirements.

Static Requirements Usually, static requirements do not influence the behavior of the components directly, but these requirements impact the embedding environment.

Refinement of the system requirements defines component requirements. These requirements apply to logic or physical components. Fulfillment of the static component-level requirements has different aspects:

- Checking the embedding system (platform) of the component is mandatory. The key questions are: What are the provided properties and guarantees of the platform and what kind of restrictions they have? Furthermore, how the other components influence the properties. Every used technology in the system narrows the design space. For example, the customer would like to install a new CCTV (Closed-circuit television is the use of video cameras to transmit a signal to a specific place, on a limited set of monitors, e.g., traffic monitoring) device, that requires high-speed throughput, but the current configuration of the embedding system provides only medium speed throughput. The platform restricts the opportunity of the expandability and trivial reconfiguration. This problem involves a design-space exploration problem at design-time;
- There are static requirements at application-level too. For example, the computation logic is unable to handle negative numbers. If the input of the component is a

negative number, the system should turn off. Usually, these application-level static requirements restrict the interfaces of the components.

In the first case, solving a design-space exploration problem could serve as a solution. It is possible to define the constraints in the system using a constraint satisfaction problem. This approach provides the opportunity to check the integrity of the system in design-time and reconfigure it in runtime. Note that, runtime reconfiguration requires to have information about the live (actual) configuration and the architecture of the system.

Checking the interface constraints in both design and run-time is possible. Using the constraint satisfaction problem extended with interface filters helps to detect violations.

Dynamic Requirements Dynamic requirements define behavioral information about the component and the system. Providing the services defined by dynamic requirements needs the guarantees of from embedding environment. The selection of the used technology influences the behavior of the components (e.g., if the embedding environment does not provide the required interfaces for the component then the operation of the component could be wrong).

Usually, dynamic requirements are defined by input-output invariants of the components. The input-output invariants are the bases of the verification of the behavior of the system. Functional interfaces describe the interfaces between the embedding platform and the component. Note that, when operation seems correct (using only several interfaces), it does not involves that the component as a whole is correct.

Fulfilling the dynamic requirements is critical for the proper operation of the system. This means the verification of the behavior is crucial.

Chapter 7 and 8 present an assume-guarantee styled design-time and runtime verification method for dynamic requirements.

Satisfying the dependability aspects of a complex CPS requires to fulfill both the static and dynamic requirements of the system.

2.4 Dependability and Security

Dependability [12] is a concept that subsumes the following attributes[30][11] (Figure 2.4):

- **Reliability -** Readiness for correct service, defined as the probability of a system or system element performing its intended function under stated conditions without failure for a given period
- **Maintainability -** Ability to undergo modifications, and repairs, defined as the probability that a system or system element can be repaired in a defined environment within a specified period of time. Increased maintainability implies shorter repair times.
- **Availability -** Probability that a repairable system or system element is operational at a given point in time under a given set of environmental conditions. Availability depends on reliability and maintainability and is discussed in detail later in this topic.
- **Integrity -** Degree to which a system or component prevents unauthorized access to, or modification of, computer programs or data.

- **Safety -** The expectation that a system does not (under defined conditions) lead to a state in which human life, health, property, or the environment is endangered.

  Two alternate definition are used [12], each focusing on different aspects.

  - ***Justification of trust****: the ability to deliver service that can justifiably be trusted.*
  - ***Decidability****: the dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable.*

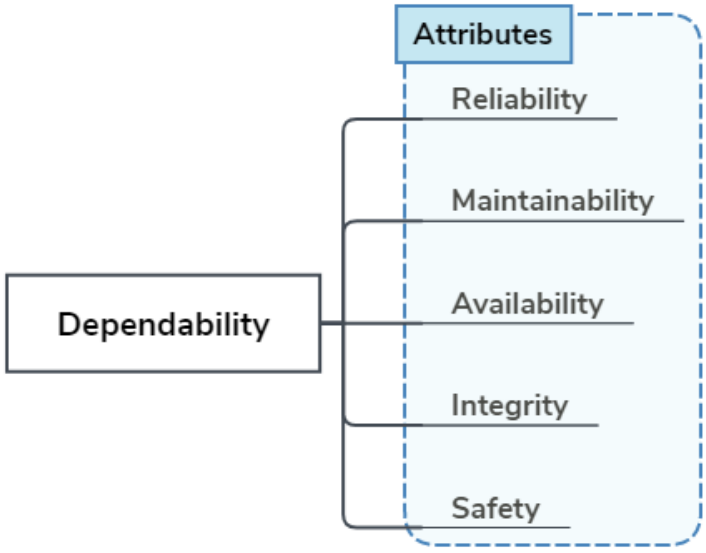

Figure 2.4: Dependability Tree

Security extends the list of attributes with confidentiality. Confidentiality, in the context of computer systems, allows authorized users to access sensitive and protected data. CPS Framework Middlewares provide built-in configuration properties to fine-tune the security aspects.

The report does not deal with the security aspects of cyber-physical systems. However, test-driven development (TDD) focusing on the security aspect as a critical part of the Industrial Internet of Thing systems [49].

2.5 Verification and Validation

The V-Model (Figure 2.5) defines the relationships between each phase of the development life cycle (refinement of the system) and the associated verification and validation (testing) phase. The horizontal axe represents the project completeness, and the vertical axe represents the level of the abstraction.

Verification and validation methods are used to check that a system meets its requirements and specifications and that it fulfills its intended purpose.

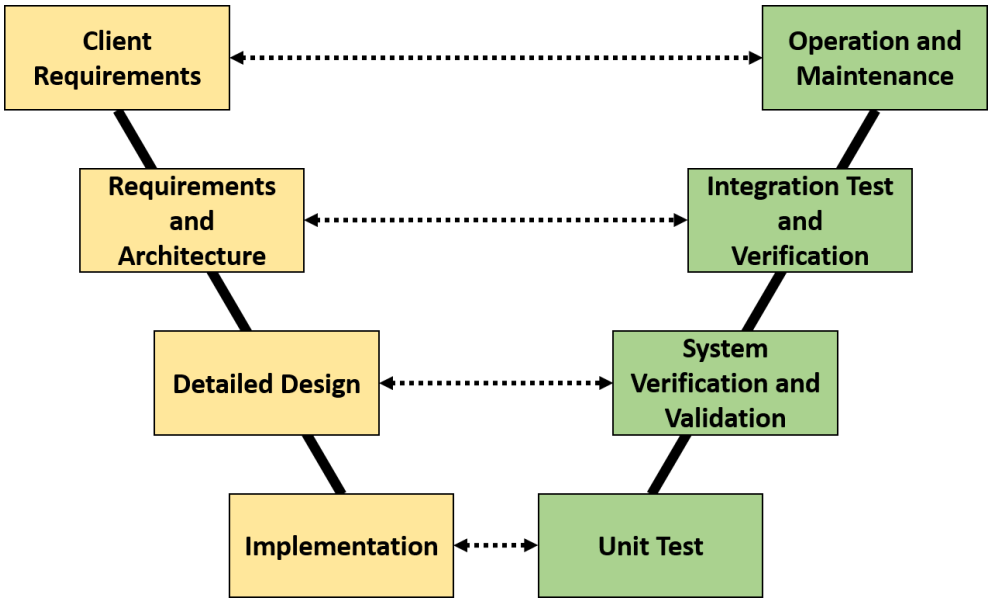

Figure 2.5: V-Model

Model-based testing Testing [53] aims to show that the actual behaviors of a system differ from the intended. The goal is failure detection in the system, so finding the differences between the implementation and the system expressed by its requirements.

Model-based testing (MBT) (Figure 2.6) is a type of testing that relies on behavior models of the planned system.

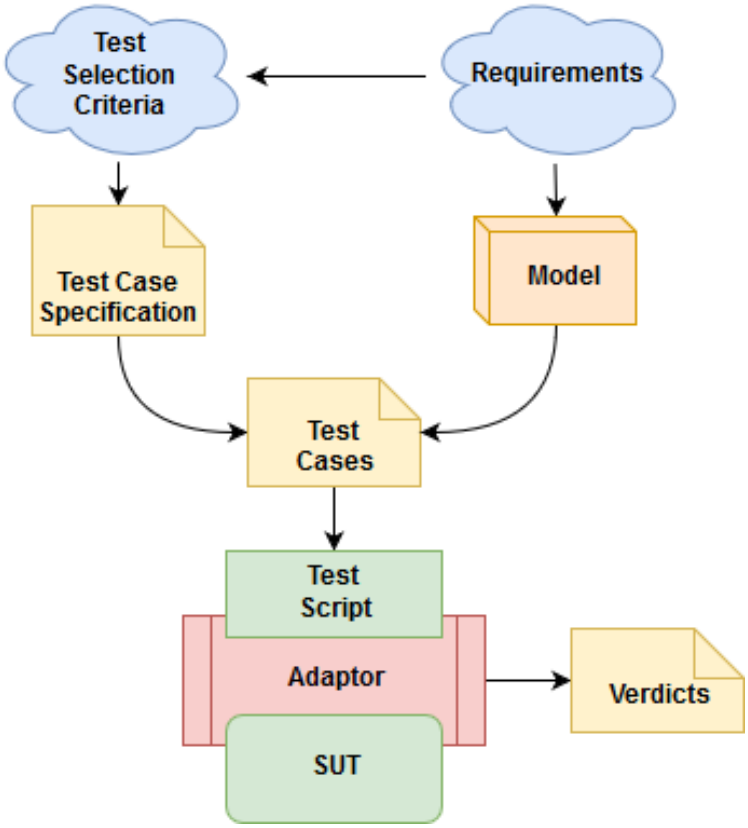

Figure 2.6: Model-based testing

The model can translate to a finite state automaton or a state transition system, representing the possible configurations of the system under test (SUT).

Model checking is a type of model verification techniques. Model checking is a technology for verifying properties of a system.

A model checker can automatically verify a property against the model if the requirements are expressed as a temporal logic formula.

2.5.1 Static Verification

Design space exploration is a popular automatic formal verification technique. These methods prove the correctness of properties by using temporal logic expressions or constraints. These methods explore the whole design space to check the required properties. It is similar to model checking paradigms that executes all possible execution path to verify a property. However, model checking exhaustive, it has problems with scalability and infinite (extra-large state space) state systems (state-explosion problem).

2.5.2 Dynamic Verification

Dynamic verification methods, like testing, are unable to deliver proper fault coverage. Testing involves the execution of a component to evaluate one or more properties. Testing delivers the test output of execution and compares it to the required output to validate the test objective.

Chapter 3

Requirement-Driven System Design Integration

This chapter introduces reference architectures and CPS frameworks in general. Frameworks correspond to best industrial practices supporting the dependability of system services. Just all major companies (e.g., Intel[1], IBM[2], and Microsoft[3]) invloved in IoT-based (Internet of Things) services created similar frameworks.

The chapter also presents CPS framework middlewares that support the architecture design and the CPS data models.

3.1 Reference Architectures

Beside the requirements, ensuring the dependability in reactive cyber-physical systems requires the proper construction of the design too. The integration and interoperability of the system need to be corroborated by the right system architecture. Reference architectures facilitate the design of the architecture. Furthermore, they facilitate the communication between the stakeholders and system-engineers by providing common terminologies and methods.

Reference architectures can provide general guidance for creating reusable general-purpose and industrial-use architectures. They assure the independence of the implementation and manufacturing neutrality. They can make suggestions about the usage of the components.

Reference architectures also give guidance to users (customers, developers) to clear the opportunities and benefits offered by cloud computing and also facilitates the communication between the customer and the cloud infrastructure provider. Reference architectures provide a general guideline in this rapidly changing technology environment.

CPS architectures generally describe a layered structure where the services of the system are implemented. The components are located in some parts of the cloud infrastructure originates in the layer specifications.

The architecture usually divided into three layers[40]:

[1] Intel® IoT Platform Reference Architecture
[2] IBM Cloud IoT
[3] Azure IoT Reference Architecture

- **Edge:** The outer layer of the architecture. Usually, it contains resource-limited devices like sensors (medical tools, consumption meters, data from mobile devices, data provided by cars), actuators (security devices, physical actuators) and target devices (crossroad lights, displays, garbage collectors). Typically, these devices use M2M (Machine-to-Machine) communication. It is advisable to control real-time devices within the Edge layer to avoid the high latency.
- **Fog:** The middle layer of the architecture, where data processing can begin. High computation power and the close connection to the Edge layer provide low latency and fast processing time. The logic components of the layer can accomplish the preprocessing of the data from sensors. Data filtering can help reduce network load. Fog layer routers connect the system to the public Internet.
- **Cloud:** The Cloud located at the top of the architecture. The layer provides the opportunity to continue processing and storing the data. The central storage facility provides reliable and secure data access for the lower (Fog and Edge) layers. The layer handles the logic for the user and business applications and the logic for managing and identifying the devices.

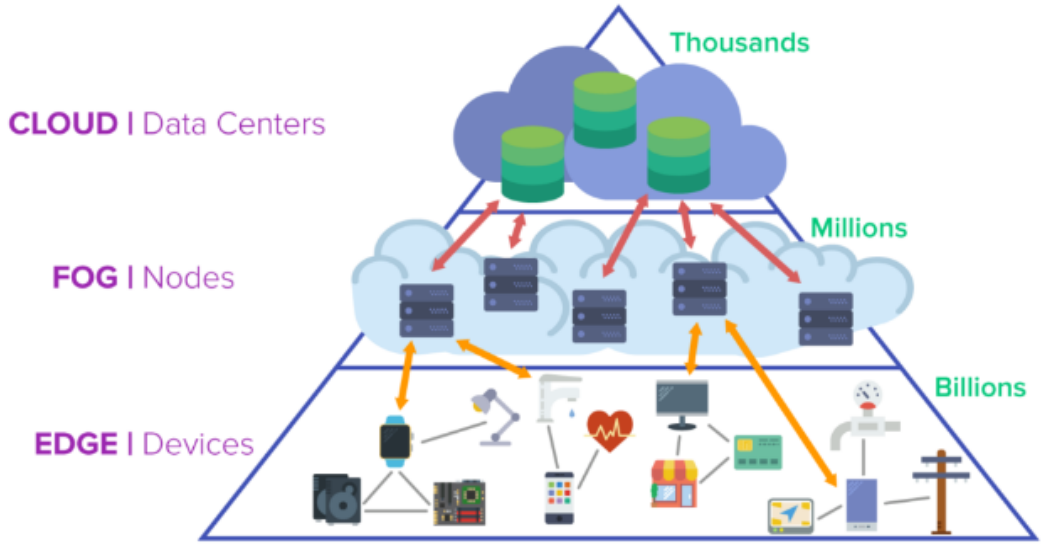

Figure 3.1: CPS Architecture Layers[1]

3.1.1 Architecture Descriptions of Systems and Software

Industrial Internet Consortium used ISO/IEC/IEEE42010:2011[33] architecture specification to define its Industrial Internet Architecture Framework (IIAF). It uses the general concepts and constructs from the standard. The IIAF is the foundation of Industrial Internet Reference Architecture (IIRA).

Based on the ISO/IEC/IEEE42010:2011 standard and the reference architecture (Figure 3.2), the Stakeholders who have interests in the System define Concerns. Concerns could have different aspects (e.g., (system) purpose, functionality, structure, behavior, cost, supportability, safety, interoperability). Some use-cases require the ability to work through different domains (e.g., healthcare with smart cities). The aspects from Concerns could

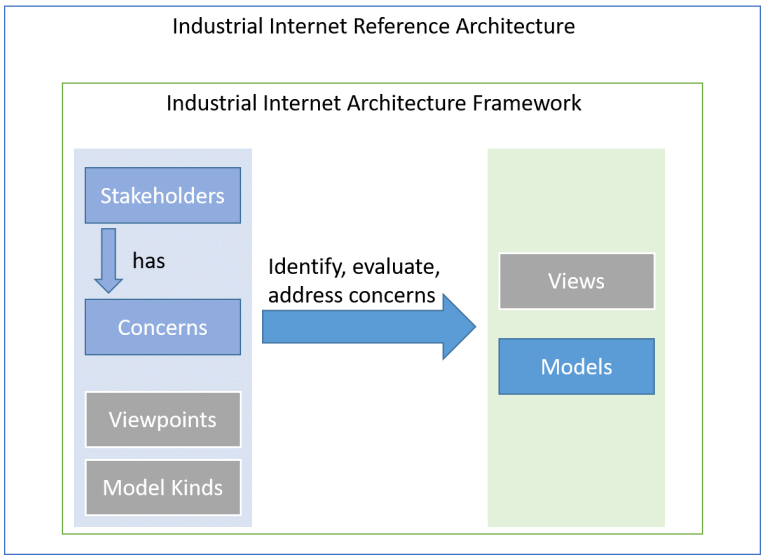

Figure 3.2: Industrial Internet Reference Architecture

appear as intra-domain extra-functional requirements (e.g., reliability, safety, security, timeliness, usability) which necessitate a semantic fusion of them.

Describe and analyze the concerns provides guidance to create abstract architecture representations. Different kind of models (behavioral, component, use case) represent the abstract view.

3.1.2 Adaption of OpenFog Reference Architecture

IEEE 1934 - Standard for Adoption of OpenFog Reference Architecture for Fog Computing The OpenFog[9] reference architecture is governed by eight (Figure: 3.3) principles. These determine the essential features of the system. The architecture is characterized by a horizontal approach.

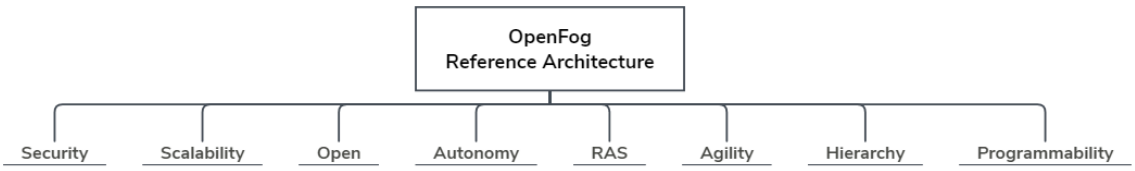

Figure 3.3: OpenFog Reference Architecture - 8 principles

It provides a variety of architectural approaches (Figure: 3.4) rely on available tools and customer requirements.

Sometimes, using cloud infrastructure is impossible (no physical access to the public Internet, fast data processing is required) or denied (regulations, data security, military usage). These exceptions may occur at different architecture levels:

- **Monitoring and Control:** Processing time-sensitive data.
- **Operational Support:** Processes, where operation is critical.
- **Business Support:** Data security restrictions.

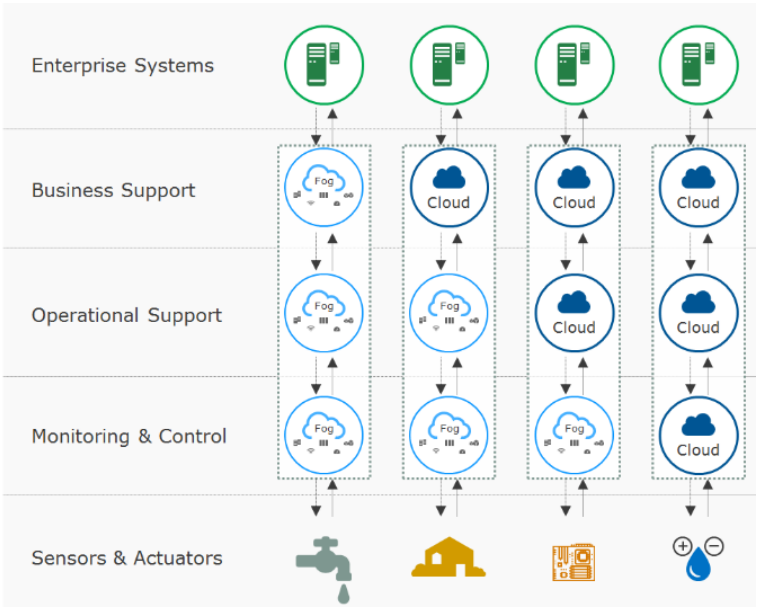

Figure 3.4: OpenFog Reference Architecture - usage model[9]

Basically, the fog architecture is not suitable for critical use cases due to the uncertainty (the properties of the fog services are unpredictable (e.g., latency, security aspects))

However, the Figure 3.4 describes the location of the component based on the provided services. This report adopts the allocation reference to determine the location of the component. Instead of the Fog services uses traditional concepts (e.g., local computing).

3.2 CPS Frameworks

3.2.1 Elaboration of System Requirements

The NIST CPS Framework [26] developed by the National Institute of Standards and Technology (NIST) is designed to support the structured elaboration of the system requirements that are the basis of the implementation of a complex and multifaceted task.

The framework defines an analytical methodology for CPS by using approaches and aspects defined in the framework documents.

It helps to identify the functionalities in a requirement-based way in different domains. Experts could collect and group the requirements in aspect groups (e.g., functional, business, trustworthiness, human, timing).

This aspect-based approach is highlighted throughout the whole design life-cycle, regardless of the used development approach (e.g., waterfall model, agile or iterative methods).

3.2.2 Adopting Cloud Computing

The Cloud Standards Customer Council (CSCC)[2] aims to accelerate the adoption of cloud computing, to standardize moving to the cloud and to create and review standards for security and interoperability issues.

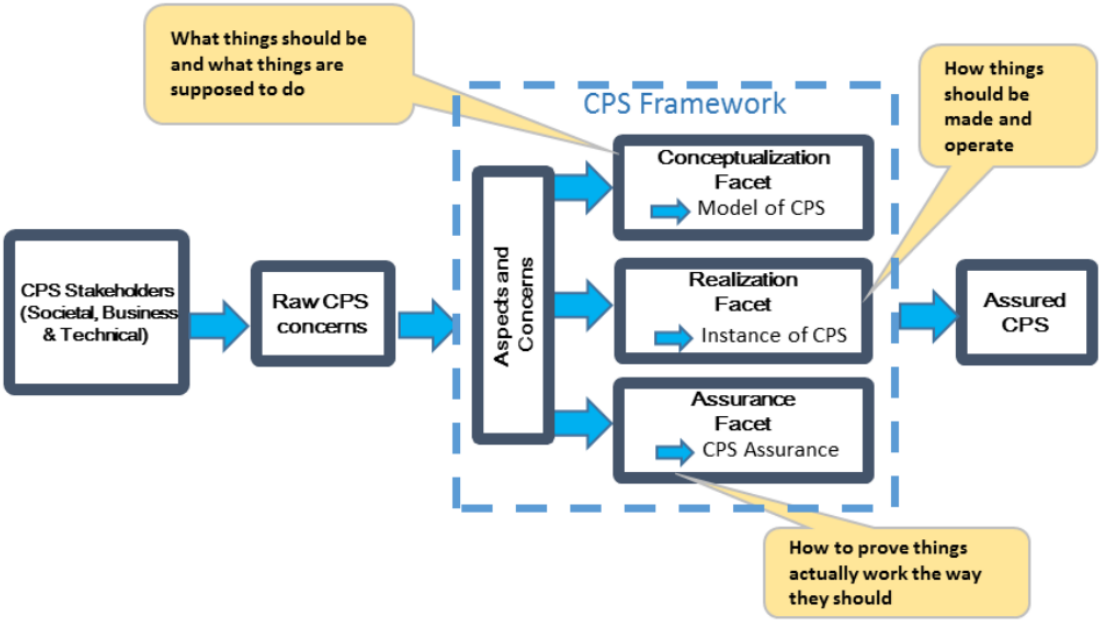

Figure 3.5: Using the NIST CPS Framework[26]

The Cloud Customer Architecture for IoT gives a clear picture of the architecture to the customer that ensures that the customer can request the proper services.

The reference architecture also provides a common foundation for the customer and the developer, delivering a general picture of the architecture, thus facilitating communication between the parties and providing a better understanding of the needs and services.

The components are classified into three architectural levels (Figure: 3.6):

- **Edge** (User Layer, Proximity Network): Sensors and user control applications are located in this layer. There is a possibility to process the data locally, thus ensuring fast response time. There may be data in the system that is denied to share on the public Internet. This secured data can also be stored and processed here.
- **Provider Cloud** (Public Network, Provider Cloud): The layer serves as a basis for cloud computing and provides switching services. The services provide high computational power. It may include asset management services and information about the component in the whole system. It can also be used to manage the IoT devices, and it is also appropriate for the pre-processing data for data visualization, which provides a more convenient analysis to the end users.
- **Enterprise Network**: The business network includes authorized user information and metadata for the system architecture. Furthermore, the collected data (both raw and processed), logging mechanism, business and management applications could be placed here.

The Cloud Standards Customer Council is now part of the Object Management Group (OMG) as the Cloud Working Group.

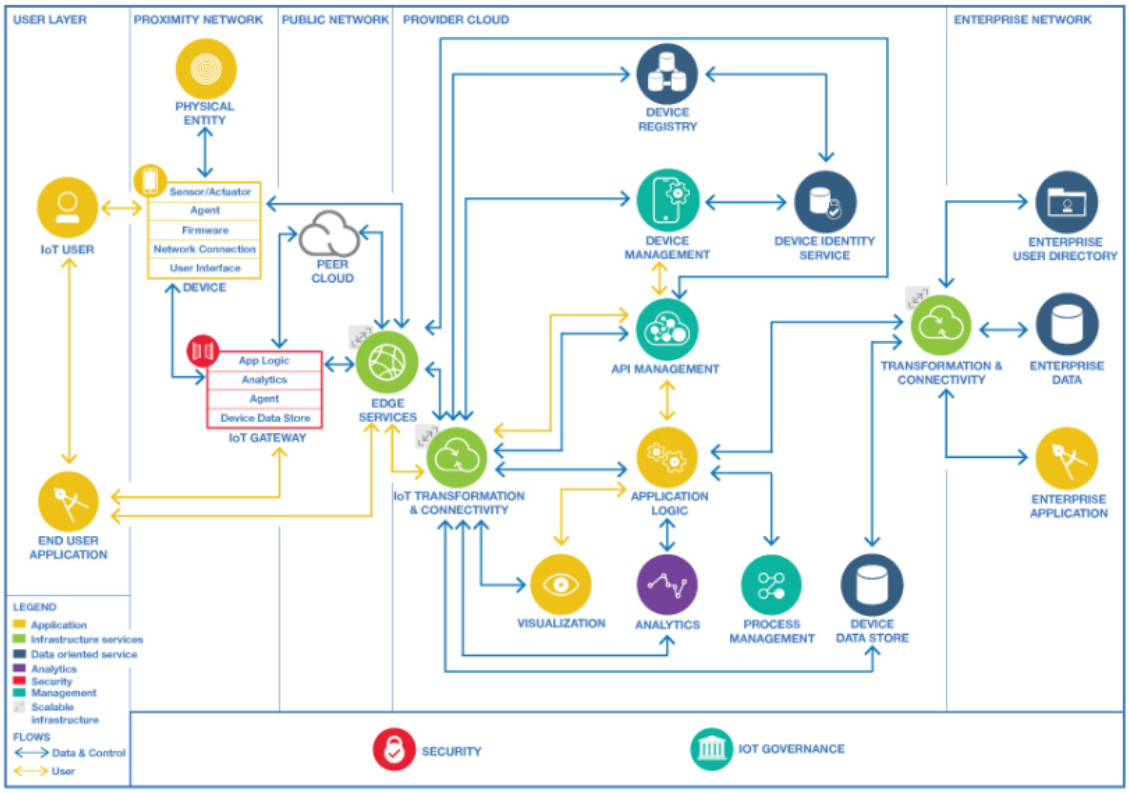

Figure 3.6: Cloud Customer Architecture for IoT[2]

3.3 Integration Requirements

As the special CPS requirements showed, ensuring the integrity is critical in CPS. The design of the system influences the functionality of the integrated system. The integrity must be ensured besides the requirements of the system. Extendability, integration, and interoperability is a key point in complex, component-based systems.

3.3.1 Layered Databus Architecture

Nowadays, smart sensor systems spread across different domains like smart healthcare, communication, transportation, manufacture, and such. This emerging system is called the Industrial Internet[4].

The security and timing requirement is much higher in Industrial Internet system than usual. Industrial Internet requires higher standards of safety and security too. Industrial Internet Consortium (IIC) with companies from several different industries have created a general reference architecture for the challenges of the Industrial Internet.

Industrial Internet Reference Architecture (IIRA) helps to achieve interoperability, provides technology guidance, and advances the development of standards. The architecture based on the Layered Databus Architecture (LDA[5]) (Databus = Data-Oriented Information Sharing Technology, which realizes the data exchange between applications through a virtual global data space) that quickly spread across the industry. LDA ensures low la-

[4]https://rti.wistia.com/medias/8ma88ry3mw

[5]http://blogs.rti.com/2017/01/31/2nd-version-of-the-industrial-internet-reference-architecture-is-out-with-layered-databus

tency, security, and direct communication between applications and domains. LDA based on the publish-subscribe communication pattern.

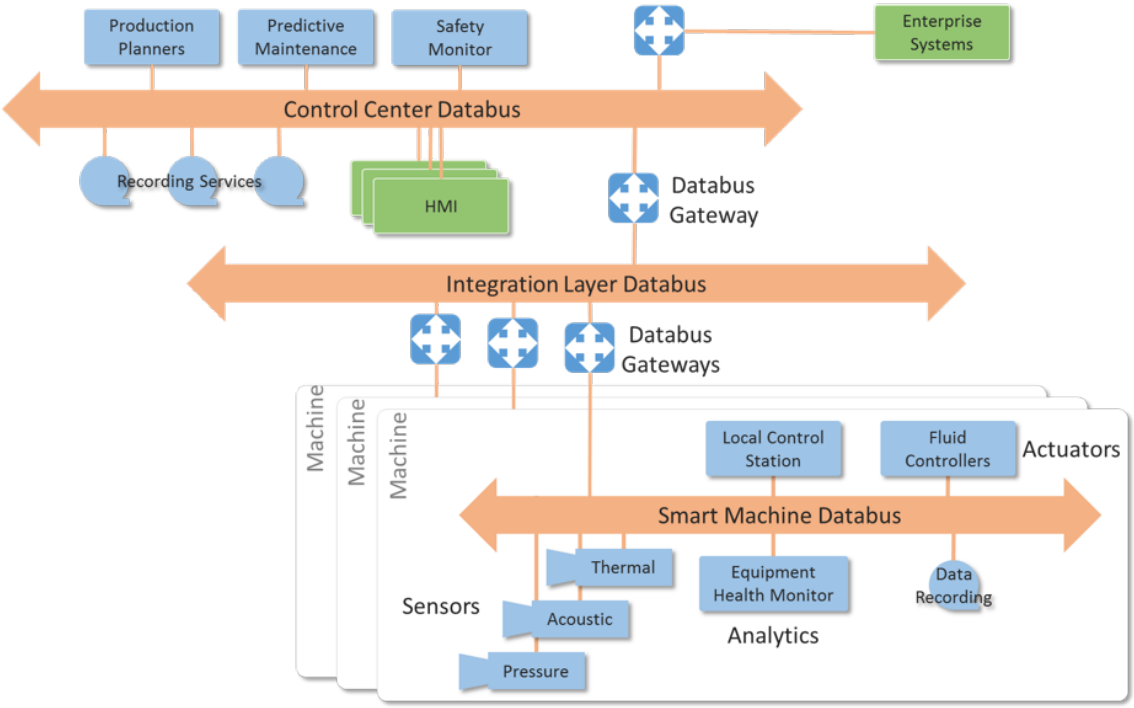

Figure 3.7: Layered Databus Architecture[7]

Between the architecture layers, gateways (= component that can be used to interconnect different networks.) ensure the correct exchange of the messages.

At the lowest architectural level, devices with low intelligence are located, which can be sensors or switches.

Sensors and actuators with low intelligence located at the lowest architectural level. Components with time-critical functionality (e.g., local control unit, healthcare monitoring tools) should be placed here because it allows quick data exchange.

Storing critical data locally (at the lowest level) also enables fulfilling data protection aspects.

Layer-to-layer communication can also be encrypted using either wired or wireless communication channel.

The highest architecture level contains the component with more complex functionality (e.g., global control unit, high-level analytics, logging services).

Layered Databus Architecture has the following advantages:

- Fast data transfer between devices
- Fast data and application discovery
- Scalable integration
- High availability and resilience
- Complex system design

The structure and the advantages of the LDA enable to use it developing complex CPS design.

3.3.2 CPS Framework Middlewares

Selecting the proper technology is a key step assuring compliance with the QoS requirement. Different technologies provide similar functionalities at a different level of extra-functional properties. Modern CPS framework technologies support a variety of run-time platforms to be general purpose ones ranging from FPGAs to server computers.

Experts have to consider all the different aspects to select the technologies for the system. It is worth to select the technologies to cover all the functional and extra-functional requirements completely.

CPS Framework middleware supports the interconnection of heterogeneous components and systems (SoS). Their goal is to provide a standardized interface for the integrity and interoperability aspects, and services for the QoS requirements.

3.3.2.1 Data Distribution Service

The Data-Distribution Service (DDS) [28] for real-time systems is an OMG (Object Management Group) standard. This middleware standard directly addresses publish-subscribe communications for real-time and embedded systems.

The DDS specifies a highly flexible, easily extensible and scalable Data-Centric Publish-Subscribe (DCPS) model for distributed communication and complex CPS application integration. The specification defines

- The Application Interface (**API**) and
- Communication Semantics (behavior and **Quality of Service**).

The main building blocks (Figure 3.8) of the DDS are the following:

- Publisher
- DataWriter
- Subscriber
- DataReader
- Topic/FilteredTopic

Publishers handles DataWriters and Subscribers the DataReaders. The communication link between the publisher and the subscriber is determined by Topics. The Topic also describes the type and the structure of the data. FilteredTopic allows to filter data, to ensure that every participant gets only the required information.

The mentioned components are linked to a DomainParticipant and DomainParticipations are linked to a Domain. Only those components that connected to the same Domain are able to communicate.

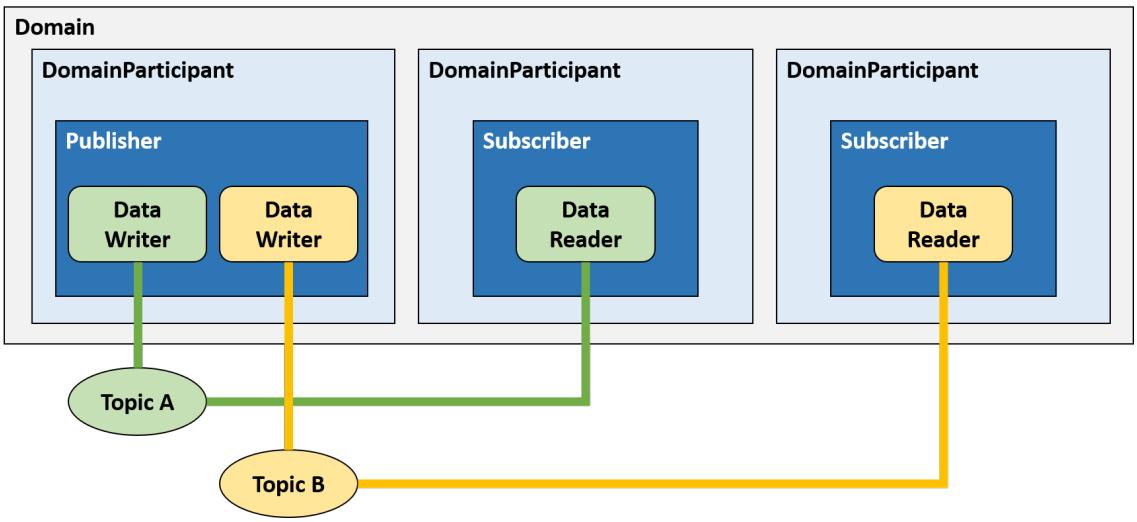

Figure 3.8: DDS Building Blocks

The quality of service [48] model provides extensive functionality to configure the system and to assure extra-functional properties. The standard defines 12 basic QoS properties, for example:

- **Deadline**: maximum time between writing/receiving new values of an instance
- **Durability**: controls whether or not, and how, published DDS samples are stored by the DataWriter application for DataReaders that are found after the DDS samples were initially written
- **History**: number of samples that DDS will store locally
- **Liveliness**: specifies how DDS determines whether a DataWriter is online
- **Ownership**: specifies whether a DataReader receive data for an instance of a Topic sent by multiple DataWriters
- **Reliability**: determines whether or not data published by a DataWriter will be reliably delivered by DDS to matching DataReaders

Several implementations are available (e.g., TwinOaks CoreDX [6] – Leading small footprint implementation, RTI Connext [7] – Full DDS implementation).

3.3.2.2 OPC UA

The OPC Unified Architecture (UA) is a platform-independent service-oriented architecture. This machine-to-machine (*M2M*) multi-layered framework accomplishes the original OPC Classic specifications goals of:

- **Functional equivalence:** all COM OPC Classic specifications are mapped to UA
- **Platform independence:** from an embedded micro-controller to cloud-based infrastructure

[6] http://www.twinoakscomputing.com/coredx
[7] https://www.rti.com/products/dds

- **Secure:** encryption, authentication, and auditing
- **Extensible:** ability to add new features without affecting existing applications
- **Comprehensive information modeling:** for defining complex information

Functional independence covers the following functionalities[8]:

- **Discovery:** find the availability of OPC Servers on local PCs and/or networks
- **Address space:** all data is represented hierarchically (e.g. files and folders) allowing for simple and complex structures to be discovered and utilized by OPC Clients
- **On-demand:** read and write data/information based on access-permissions
- **Subscriptions:** monitor data/information and report-by-exception when values change based on a client's criteria
- Events: notify important information rely on client's criteria
- **Methods:** clients can execute programs, etc. methods defined on the server

Originally the OPC UA standard supports the Client-Server architecture however since the Pub-Sub-Specification the Publish-Subscribe model is also supported.

Figure 3.9 shows the simplest OPC Client-Server architecture. The OPC Server directly to the digital field (edge) devices. Usually, OPC client processes the data and connects to a SCADA system.

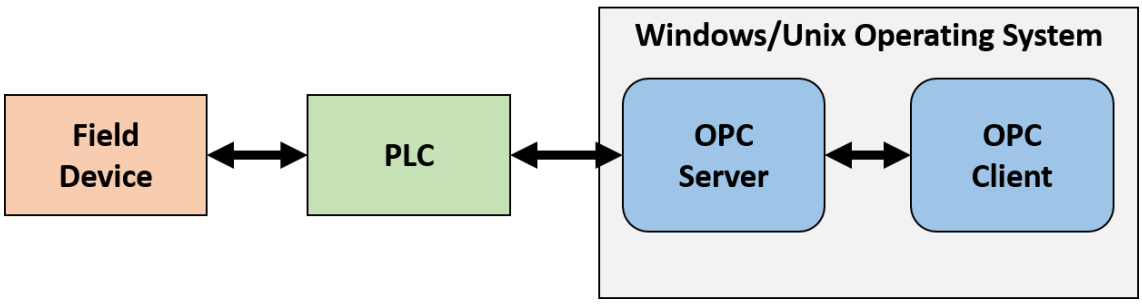

Figure 3.9: OPC UA Client-Server Architecture

3.3.2.3 MQTT

MQTT [15] is an OASIS standard. It was designed to minimalize network bandwidth and the resource requirements. It is ideal for a lightweight integration of connected devices and mobile applications.

In MQTT two main components are distinguishable:

- **Client**: Device or application that MQTT library. Clients are connected to brokers.
- **Broker**: The broker receives all messages, filters the messages, determines who is subscribed to each message, and sends the message to these subscribed clients.

[8] https://opcfoundation.org/about/opc-technologies/opc-ua/

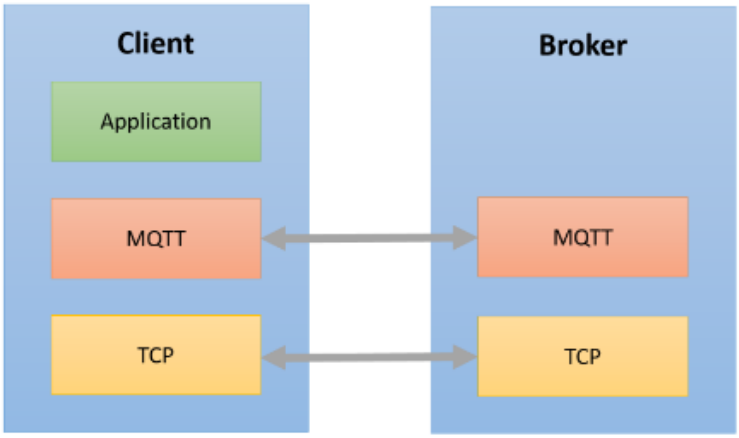

Figure 3.10: MQTT Archtecture

After building the connection with the Broker, Clients can subscribe to Topics or they can publish data (Application Message) to subscribed Clients. There is also an option to unsubscribe from Topics.

3.3.2.4 Heterogeneous Solutions

It is often required in complex CPSs that satisfying the wide range of requirements needs many different technologies.

In general, interoperability within the technologies is determined, but there is no guarantee between them.

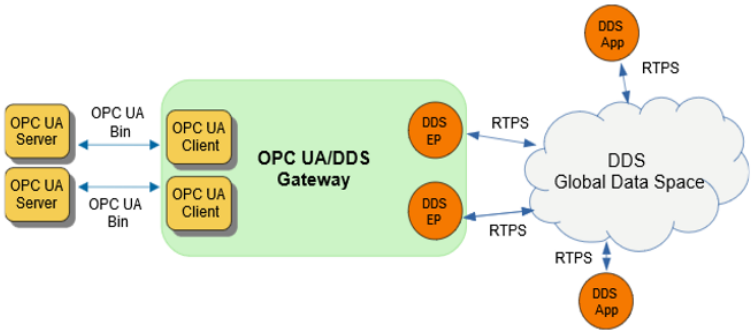

Figure 3.11: OPC UA/DDS Gateway[4]

Figure 3.11 shows the OPC UA/DDS Gateway that enables DDS and OPC UA applications to interoperate transparently using the native mechanisms and APIs of each specific framework.

Gateways usually perform the conversions (e.g., message format, encoding) between different technologies.

The section about the reference architectures also showed that gateways are often an indispensable part of a CPS architecture.

Chapter 4

Component and System Modeling

This chapter is about CPS component and system modeling using and adoption of model-driven development[44].

A simplified adaption of the classic modern-driven development approach was used in this report. Using CPS Framework middlewares provide a dataflow oriented approach to design the system.

The main advantage of the middlewares that they provide data-driven paradigm. Ad-hoc and spaghetti paradigms are not acceptable in case of safety-critical CPSs.

The architectural clarity of CPS means that CPS engineering uses specific SysML (or UML) diagrams without the ordinary diagrams. The general architecture (e.g., reference architectures) involves the reduction of the number of the diagrams. The simplified modeling requires simpler mathematical models to represent the system. This chapter describer mathematical formalisms to model a CPS.

The gradual refinement of the use case and activity diagrams classifies the type of models into three main categories: behavioral, structure, and platform. System services and the configuration parameters define the platform model. Classical structures models are available for CPS components, but the integration models integration model use a higher abstraction.

The most common behavioral diagram is the Dataflow diagram. The data-driven paradigm of CPS involves the key role of the diagram. (e.g., RTI provides a dataflow-based lightweight DDS modeling tool)

It is possible to model the structure and platform models with SysML Block Definition Diagram (BDD) and Internal Block Diagram (IBD).

4.1 Adaptation of Model-Driven Engineering

Model-driven engineering is a software and system design paradigm for the development of component-based systems. It structures the specification and the specification based requirements in the form of models. Object Management Group (OMG) started to standardize the Model-Driven Architecture (MDA) approach in 2001.

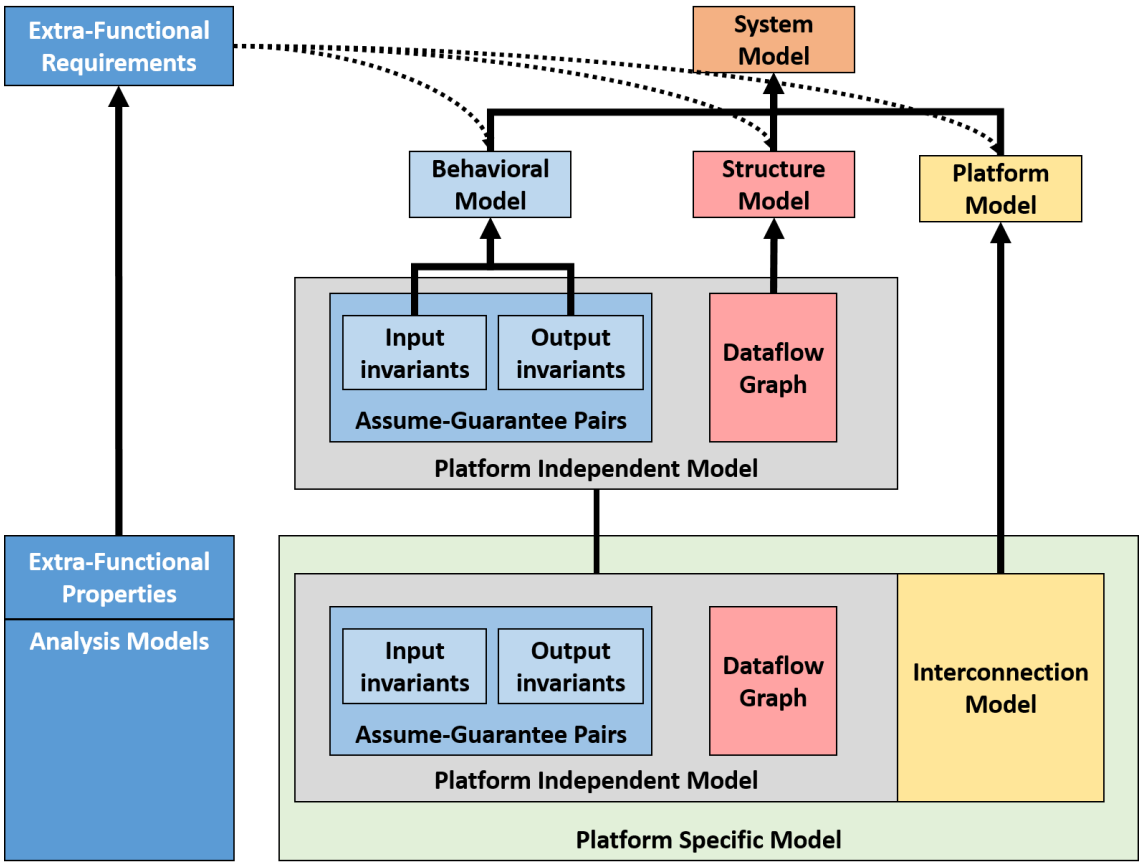

Figure 4.1: Model-Driven Engineering

4.1.1 Platform Independent Model

The model-driven architecture approach defines system functionality using a *platform-independent model* (PIM). PIM is a model of a system that is independent of the specific technologies used to implement it.

The construction of high-level system models begins with the analysis of the functional and extra-functional requirements. Static and dynamic models are distinguishable from the beginning of the design process.

Usually, the functional decomposition of a system defines both static and dynamic requirements. However, most of the extra-functional requirements are static There are dynamic requirements too (e.g., resilience aspect, temporal-like).

Typically, static models are responsible for the structure (architecture, composition, components) and platform (deployment, integration) models and dynamic models cover the behavior (functional, logic, interaction) of the system.

In this report, different mathematical formalisms were used to represent the system models. The verification methods reuse the assume-guarantee approach that provides the opportunity to check both the static and dynamic models and perform runtime verification, and on the constraint satisfaction problem to check the static models.

The assume-guarantee approach and the constraint satisfaction problem-based verification will be presented later in this report.

The platform independent model refines the requirements into the behavioral and the structural models. It is possible to define the assume-guarantee pairs in this early phase of the system development with the terms of input and output invariants. The input-output invariants also defines the components. The dataflow models represent the flow of data through the system and present a high-level overview of the system. PIM neglects the structure and the properties of the data.

As an example (Figure 4.2 - top), the PIM defines the components (Publisher, Subscriber) and the dataflow between them. This construction is modeled as an ordered graph.

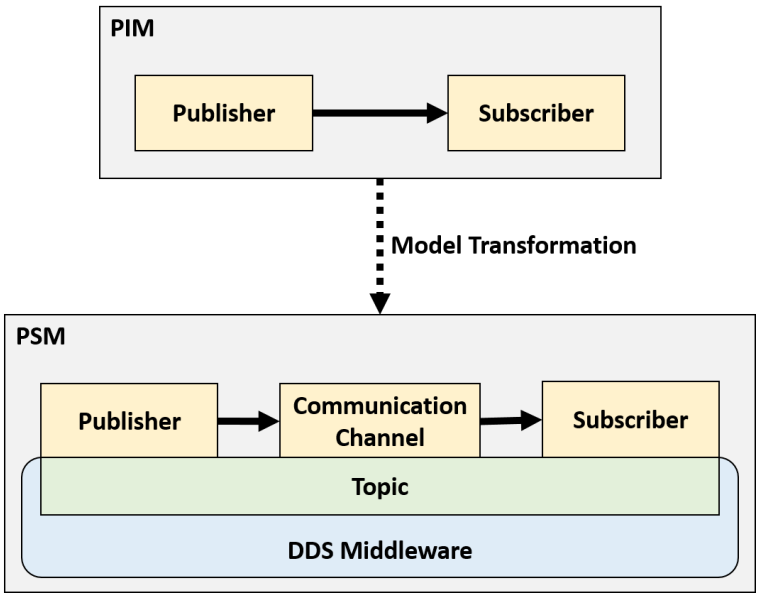

Figure 4.2: PIM and PSM Models

4.1.2 Platform Specific Model

Platform specific model is a refinement of the PIM. In the PSM, the PIM extended with the communication model. The communication models describe the properties of the communication channels and show how the components will be deployed. Furthermore, PSM specifies technology specific aspects.

The connection between the components and the platform modeled by its functional interfaces.

Figure 4.2 (bottom) shows the refinement of Figure 4.2 (top) extended with the Communication Channel (using DDS) between the Publisher and the Subscriber.

4.2 Architecture Modeling Foundations

The structure and the properties of a graph make it possible to use it to model static system architectures or dynamic behaviors. This section presents the graphs in general to provide the basis for more complex and detailed formalisms.

Graph is an ordered pair $G = (V, E)$, where V is a non-empty set and $E \subseteq V \times V$. The elements of V called vertices and the elements of E called edges.

If G is a graph then $V(G)$, and $E(G)$ denote the sets of nodes and edges. $v(G)$ and $e(G)$ denote the number of the nodes and edges.

Directed and Undirected Graph In undirected graphs the edge (x, y) is identical to the edge $(y, x,)$. In directed graphs, the values of the edges have orientations.

Simple Graph If $e = (v_1, v_2) \in E$ then (v_1, v_2) are the end points of the edge. If $v_1 = v_2$ then e is a loop. Two or more edges that connect the same two vertices are called multiple edges. A simple graph is graph without multiple edges or loops.

Bigraph is a graph (Figure 4.3) whose vertices divided into two disjoint sets V_1 and V_2.

$G = (V_1, V_2, E)$ denotes a bigraph whose partition has the parts V_1 and V_2. E denotes the edges of the graph.

Every edge connects a vertex in V_1 to one in V_2. $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$ where $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$

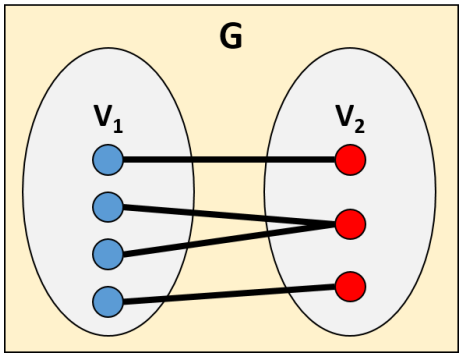

Figure 4.3: Bigraph

Note that, the PSM model of a system could correspond to a bigraph. The two sets are the components and the communication channels

4.3 Static Models

Several design question requires to extend the scope of validity of a local component property to other components or the whole system. Furthermore, checking the functional and the architecture model consistency is also required. Using the constraint satisfaction problem as a mathematical model providing the opportunity to build global constraints from local ones. In the case of a static example, the variables are the properties and the logical relations are the constraints.

4.3.1 Constraint Satisfaction Problem

Constraint Satisfaction Problem (CSP) is a mathematical problem defined as a set of variables whose state must satisfy a number of constraints.

Formally, a constraint satisfaction problem is defined as a triple $\langle X, D, C \rangle$, where

- $X = \{x_1, x_2, \ldots, x_n\}$ set of variables
- $\forall x_i : \exists D_i$ a finite set of possible values (domain of variable x_i)

– $C = \{c_1, c_2, \ldots, c_n\}$ set of constraints

$\forall c_i \in C$ constraint defines a relation over a set of $x \subseteq X$ variables.

Classification of Constraints Three type of constraints are distinguishable rely on the number of variables that are affected by the constraints:

- **Unary:** Only one variable is affected by the constraint. (Preprocessing deals with these type of constraints.)
- **Binary:** Two variables are affected bt the constraint.
- **Higher-level:** More then two variables are affected.

Higher-level constraint require binarization processes because the evaluating algorithms use binary constraints.

Completeness $\mathbb{S}$ is a set of all proper evaluations and solutions. An evaluation $\mathbb{S}$ is consistent and complete if it accepts all of the constraints and includes all variables.

Solutions An evaluation is a solution if it is consistent and complete:

– $\mathbb{S} = \{S_1\} : S_1 \in S$, just one solution

– $\mathbb{S} = \{S_1, S_2, \ldots, S_n\} : S_i = S$, all solutions

– $\mathbb{S} = opt(S_1, S_2, \ldots, S_n) : S_i \in S \wedge opt()$, where $opt()$ is a objective function, given an optimal (or at least good) solution

Constraint Graph The constraint graph (Figure 4.4) is a graph where nodes represent the variables, and the edges represent the constraints:

– $\forall v_i \in V \rightarrow x_i \in X$, and

– $\forall e_k \in E \rightarrow c_k \subseteq C$

Example Figure 4.5 presents a cryptarithmetic CSP example. The following equations define the constraint in the example:

1. $O + O = R + 10 \times X_1$
2. $X_1 + W + W = U + 10 \times X_2$
3. $X_2 + T + T = O + 10 \times X_3$
4. $X_3 = F$

The X_1, X_2, X_3 variables represent the transmission values of the equations. The right part in the Figure 4.5 is the constraint (hyper-)graph representation of the example. Each constraint is a square box connected to the variables it constrains. The constraint nodes in the third row of the example connected to the corresponding equation variables.

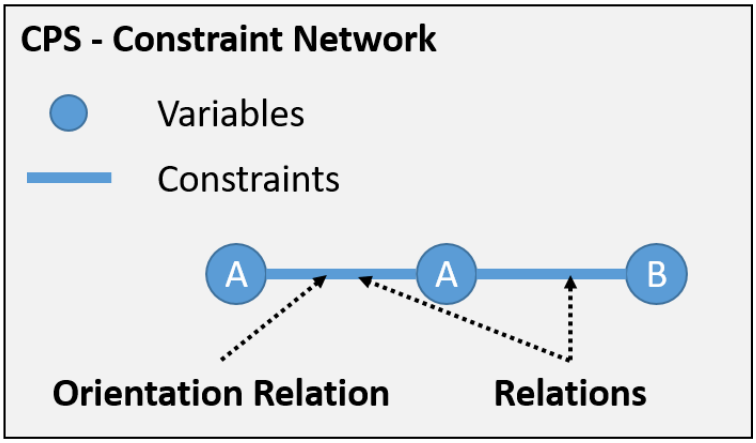

Figure 4.4: Constraint Graph

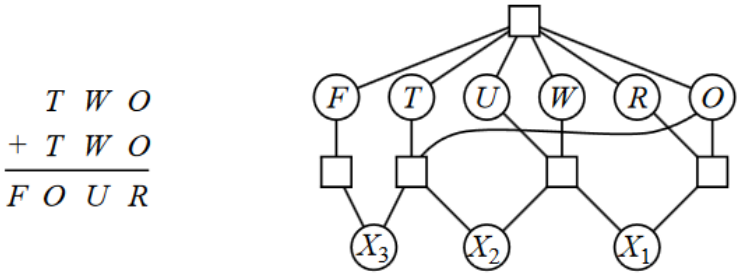

Figure 4.5: CSP Example: Cryptarithmetic

4.3.1.1 Evaluating CSP

In CSP evaluation, the first step is preprocessing. Preprocessing investigates the unary constraints for each node. By performing the preprocessing step, the domain of the variables can be significantly reduced. By reducing the domains, the efficiency and speed of CSP solver algorithms can be much effective. This step is indispensable to solve large-scale problems.

"Generate and Test" solving method systematically generates all possible variable assignments and then verifies whether all the constraints are correct for a given assignment. The method is inefficient because it contains many false assignments that are discarded during the testing phase.

Another and more efficient solving method for performing CSP evaluation is "Backtracking". After choosing a value for a variable, the algorithm immediately checks the consistency of the partial solution. If there is no acceptable value for a variable, the algorithm changes the value for previously assigned one, until possibilities are left or getting a complete solution.

Several consistency techniques (node and arc consistency) for constraint graphs are available to reduce the search space.

In node consistency, a variable x_i in the constraint graph is node consistent if $\forall d \in D_i$ (domain of x_i), each unary constraint on x_i is satisfied.

Node consistent constraint graph satisfies all unary constraints so that they can be removed from the graph. The remaining part of the graph contains binary constraints. In the constraint graph, binary constraints correspond to arcs.

Arc (x_i, x_j) is arc consistent if $\forall d \in D_i$ (domain of x_i), there $\exists y \in D_j$ (domain of x_j) such that $x_i = d$ and $x_j = y$ is permitted by the binary constraint between x_i and x_j. If an arc (x_i, x_j) is consistent, than it does not automatically mean that (x_j, x_i) is also consistent. The constraint graph is arc consistent if all arcs are arc consistent from both directions.

Arc consistency algorithms are available (e.g., AC-1, AC-3, AC-4) [37] [41] with different efficiency and algorithmic method.

4.3.1.2 Dynamic CSP

It is not always possible the use of CSP in a continuously changing environment. Dynamic CSP (DCSP) [20] is a solution to this problem (e.g., a set of constraints evolving because of the environment). DSCP is a sequence of CSPs. Each CSP is a transformation of the previous one. The transformation contains the addition or removal of the variables and the constraints. The solving method can be classified into three categories: Oracles, Local repair, Constraint recording.

The report does not deal with the DCSP in details but there is a possibility to use DCSP in a continuously changing CSP. It provides a dynamic and efficient solving method to check the changing system constraints.

4.4 Dynamic Models

4.4.1 Labeled Transition System

Labeled Transition Systems used to describe the behavior of discrete systems. LTS consists of states and transitions between states, which labeled with observable $\mathcal{A}ct$ and unobservable τ actions. The report uses the definition from [16]

Mathematically, an LTS is a 4-tuple $M = \langle Q, \alpha M, \delta, q0 \rangle$, where:

- Q is a non-empty set of states
- $\alpha M \subseteq \mathcal{A}ct$ is a finite set of observable actions (alphabet of M)
- $\delta \subseteq Q \times (\alpha M \cup \{\tau\}) \times Q$ is a transition
- $q0 \in Q$ is the initial state

Let $M = \langle Q, \alpha M, \delta, q0 \rangle$ and $M' = \langle Q', \alpha M', \delta', q0' \rangle$. M transits into M' with action a, denoted $M \xrightarrow{a} M'$, if and only if $(q0, a, q0') \in \delta$ and $\alpha M = \alpha M'$ and $\alpha M = \alpha M'$.

Traces A trace t of an LTS M is a sequence of observable actions that M can perform at its initial state. $t \restriction \sum$ denotes the trace obtained by removing from t all occurrences of actions $a \notin \sum$. The set of all traces of M is the language of M ($\mathcal{L}(M)$).

Let $t = \langle a_1, a_2, \ldots, a_n \rangle$ be a finite trace of LTS M. $[t]$ denotes the LTS $M_t = \langle Q, \alpha M, \delta, q0 \rangle$ with $Q = \{q_0, q_1, \ldots, q_n\}$, and $\delta = \{(q_j, a_i, q_i)\}$, where $1 \leq i \geq n$.

Parallel Composition The operator $\|$ combines the behavior of two components by synchronizing the actions.

Let $M_1 = \langle Q_1, \alpha M_1, \delta_1, q0_1 \rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q0_2 \rangle$ are two LTSs. $M_1 \parallel M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q0 \rangle$, where $Q = Q_1 \times Q_2, q0 = (q0_1, q0_2), \alpha M = \alpha M_1 \cup \alpha M_2$, and t is defined as follows:

$$\frac{M_1 \xrightarrow{a} M_1' \quad a \notin \alpha M}{M_1 || M_2 \xrightarrow{a} M_1' || M_2}$$

$$\frac{M_1 \xrightarrow{a} M_1' \quad M_2 \xrightarrow{a} M_2' \quad a \neq \tau}{M_1 || M_2 \xrightarrow{a} M_1' || M_2'}$$

The parallel composition operator is commutative and associative.

Properties and Satisfiability A property P defines the set of acceptable behaviors over . An LTS M satisfies P, denoted as $M \models P$, if and only if $\forall t \in \mathcal{L}(M).t \upharpoonright \alpha P \in \mathcal{L}(P)$

4.4.2 Finite State Machines

A finite-state machine (FSM) is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. External inputs (actions) make the FSM change from one state to another. These changes between the states are called a transition. An FSM is defined by a list of its states, its initial state, and the conditions for each transition. Two types of FSMs are distinguishable: deterministic and non-deterministic finite state machines. This report deals with deterministic finite state machines.

Acceptor FSM Acceptors (Figure 4.6) produce a binary output, indicating whether or not the received input is accepted. Each state of an FSM is either "accepting" or "rejecting".

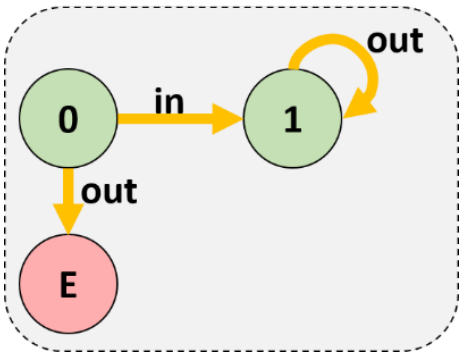

Figure 4.6: Acceptor FSM

In this methodology, the error ("rejecting") states indicate an error in the system. If one of the test sequences enters the error state, it considered as a bad trace that violates the property.

Mathematical Formalism Mathematically, a state machine is a 5-tuple: $M = \langle \sum, S, s_0, T, F \rangle$ where,

- $\sum$ is a finite set of input actions (alphabet of the machine)
- $S = \{s_1, s_2, \ldots, s_n\}$ is a non-empty, finite set of states
- $s_0 \in S$ is the initial state
- $T \subseteq (E \times S) \times (S \times O)$ is a finite set of transitions, that represent changes of states in response to input actions generate output actions
- $F \subset S$ set of final states that represent the "rejecting" states

4.4.3 Linear Temporal Logic

Linear temporal login LTL is a modal temporal logic with modalities referring to time.

LTL is built up from a finite set of propositional variables, logical operators, and temporal modal operators.

A valid LTL formula has the following syntax:

$$\varphi ::= a | \varphi_1 \wedge \varphi_2 | \varphi_1 \vee \varphi_2 | \neg\varphi | \mathbf{X}\varphi | \varphi_1 \mathbf{U} \varphi_2$$

, where:

- a is an atomic proposition
- $\varphi, \varphi_1, \varphi_2$ are valid LTL formulas
- **X** denotes the "next" operator
- **U** denotes the "until" operator

4.4.3.1 Main Operators

Atomic Proposition An atomic proposition is true on a path, if it holds on the first state of the path. Formally definition: Given a path π and an atomic proposition a

$$\pi \vDash a \equiv a \in L(\pi[0])$$

Negation of Atomic Proposition The negation of an atomic proposition holds on a path π if and only if the atomic proposition does not hold in the first state of the path:

$$\pi \vDash \neg a \equiv a \notin L(\pi[0])$$

Conjuction Given two LTL formulas φ_1 and φ_2 the conjunction holds on a path π if and only if both formulas hold on the path.

$$\pi \vDash \varphi_1 \wedge \varphi_2 \equiv \pi \vDash \varphi_1 \text{and} \pi \vDash \varphi_2$$

Next The next operator is used to specify that a formula holds *at next*, that is, at the next position in a path.

$$\pi \vDash \mathbf{X}\varphi \equiv \pi\,[1..] \equiv \varphi$$

Until The until operator specifies that a formula is true until another one is true. There are two parts in the definition of $\varphi_1 \mathbf{U} \varphi_2$

1. formula φ_1 must hold at some position on the path;
2. at all previous positions, formula φ_2 must hold.

Eventually This operator is noted **F**. $\mathbf{F}\varphi$ means that φ must hold somewhere in the future. Formally, $\mathbf{F}\varphi$ is defined as True$\mathbf{U}\varphi$.

Globally This operator is noted **G** This is the dual of the eventually operator. $\mathbf{G}\varphi$ means that φ always holds. Formal definition: $\neg\mathbf{F}\neg\varphi$

Chapter 5

Design for Dependability

5.1 Multi-Aspect Requirements in CPS

To guarantee the extra-functional properties of CPS services is essential for the proper work of the system. Mismatches between the technologies and requirements can lead to a contradiction. The as early, as possible detection of conflicts between assured properties of technologies and the requirements of the target application drastically reduces the cost and the development time.

The experts have to select the proper technology to satisfy the extra-functional requirements related to nodes and their interconnections.

A local decision on selecting a particular technology for a node or communication link can imply exclusions of other technologies in the surroundings of the system topology. These exclusions are inside and between the nodes. They are propagating through the system. It is easy to check these implications with a systematic search on a small system, but it is much harder in a complex cross-domain system.

The next section aims to give a methodology to verify the system from this aspect with constraint satisfaction problem (CSP). The methodology can provide a solution to verify the configuration and the architecture model consistency.

5.2 Static CPS Model

The design workflow starts with a widely used engineering model and performs the requirement-technology mapping through a series of derived models [25].

1. **SysML** The System Modeling Language[1] (SysML) [21] is a general-purpose modeling language for systems engineering, in particular for embedded and CPS applications thanks to its support for describing physical entities.

   As SysML is widely used to model CPS, it was a natural starting point of the methodology shown in this report. Moreover, the SysML supports multi-aspect modeling and provides a toolset to add constraints to the system.

   The fundamental building element of SysML is a Block. A Block can represent the physical, logic components or the constraints related to them.

[1] http://www.omgsysml.org/

2. **Graph model** GraphML [19] is an XML structure for representing graphs. The language contains the basic graph components, and it provides extra properties and notions to extend the functionality. It is easy to read and parse. The basic components are the Nodes, Edges, Properties (Labels). The labels are constraint variable vectors (Figure 5.1). Each node has a vector containing allocated (requirement-based) and free variables.

3. **CSP** XCSP3 [14] is a good, solver-independent format to model CSP. XCSP3 is comprehensive, easy-to-use and XML-based supporting many types of variables (e.g., Integer, Real, Symbolic, Graph,.. variables) and constraints (e.g., intension, extension, allDifferent, allEqual). A variety of sophisticated CSP solvers support the XCSP3 format.

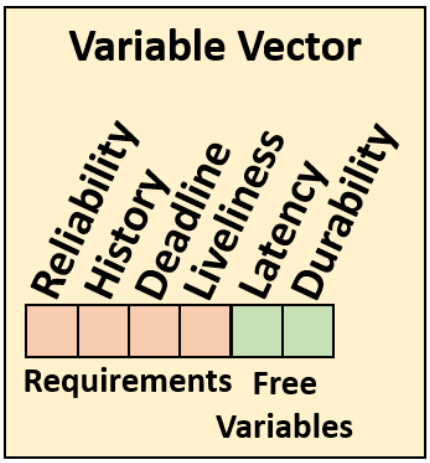

Figure 5.1: Variable Vector

5.3 Verification Method

5.3.1 Architectural Description by CSP

It is possible to formalize the conflicts in the system to build a general rule set to verify the system automatically. A transformation to a constraint satisfaction problem provides a systematic way of the solution to the satisfaction of compliance with requirements. Note, that the core idea strongly resembles the fundamental ATPGF algorithms regarding a successive try-and –error allocation of node properties. Graphs can simply represent a CSP with only unary and binary constraints like the following simple example shows:

Several solution algorithms exist for this restricted CSP class over graph representations (Figure 5.2) (e.g., AC-1, AC-3, AC-4) [37] [41]. The solution of the general class of a complex multi-variable CSP requires at first a binarization of all the non-binary constraints [13]. Subsequently, the solution algorithms mentioned above can process the reduced graph-structured CSP.

5.3.2 Defining Constraints

Shaping Constraints The very first and decisive step of CPS architecture design is the selection of the topology-implementation technology paradigm mapping. For instance, this

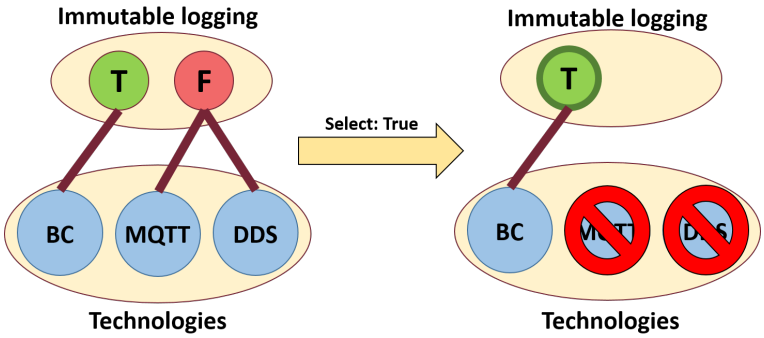

Figure 5.2: Constraint Graph

highest level decision confines the implementation of a particular communication link to a technology assuring a high-speed, but the detailed engineering dimensioning happens in a subsequent design phase. Finite, qualitative categories (e.g., throughput – high, normal, low) describe the QoS characteristics demanded by the requirements and those offered by the candidate implementation technologies. This resolution can provide a proper abstraction, and it is open to specify detailed solutions if needed.

The design constraints describe the set of appropriate implementation technologies in a declarative way. For instance, a link required to be of at least "medium" speed needs an implementation technology assuring a "high" or "medium" bandwidth, but excludes the use of a "low" speed one.

Note, that different kinds of extra-functional requirements may appear in an aspect-oriented way. The resulting solution space will consist of only the candidate technologies fulfilling all the requirements.

Complex Constraints Decomposition techniques help to refine the constraint (Figure 5.3). Design patterns could expand the functionality of the primary components, towards subsystems to fulfill complex requirements.

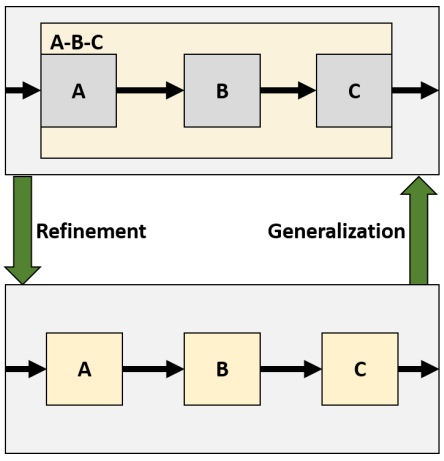

Figure 5.3: Complex Constraints

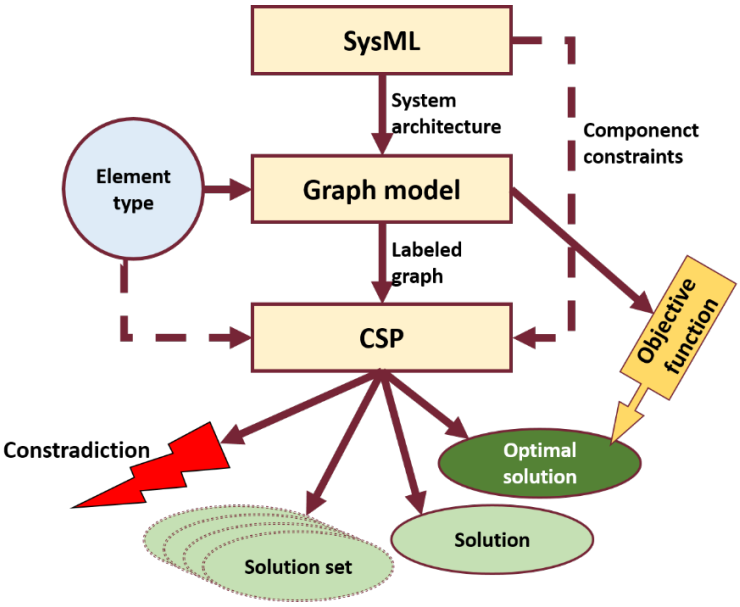

Figure 5.4: Verification Method

5.3.3 Model Transformations

Figure 5.4 shows the abstraction [54] layers of the methodology with the additional components and the possible outputs.

There are three abstraction layers. Each layer represents a view of the system architecture model. Different transformation rules act between the layers [47]. These transformations support traceability.

1. **SysML to GraphML** The transformation between SysML and GraphML is simple. The component blocks in the SysML model correspond to nodes in the graph abstraction. The labels of the nodes and edges carry the constraints connected to the SysML blocks. The GraphML representation is a high, topology oriented abstraction of the constraint graph maintain the semantic constraints as labels.
2. **GraphML to XCSP3** XCPS3 provides the functionality to solve the constraint network of the CPS. GraphML and XCSP3 are both XML types thus the conversation between them is simple with parsing. Parsing has two steps:
   (a) **Define the variables** Every required extra-functional aspects has their own domain set (e.g., Aspect: Technology – Domain: DDS MQTT; Aspect: Throughput – Domain: low, medium, high). The aspect name and do domain define a variable. Every component in the system needs its variable for all the aspects.
   (b) **Define constraints**
      i. **Constraints for the preset variables** Come from the system requirements and usually connect to an architectural block. They define every extra-functional requirement (e.g., The connection to the cloud component has high latency.).
      ii. **System constraints** Globally general constraints (e.g., The throughput should be ‘medium’ between all components.). Solvers use this XCSP3 file to solve the problem.

5.3.4 Solving the CSP

CPS solvers are applications (e.g., AbsCon[2], Choco[3], OscaR[4]) which solving CSP with CSP is solving algorithms. They usually provide an API to access to their functionalities. The quality and the efficiency of the solvers move on a wide scale. The result of the problem could be:

- **Contradiction** – The system contains contradiction(s).
- **Result set** – Every possible result.
- **One result** – A result from the result set.
- **Optimal result** – An optimal solution rely on a goal function.

Finding the optimal result is hard. The XCSP3 provide the toolset (optimization constraints) to model an optimal solution. The first XCSP3 solving competition [3] results could offer an overview of the solver's efficiency.

5.4 Method Summary

The described method provides an algorithmic verification through the whole system design space. The SysML integration is useful for real projects.

The method is independent of the technologies thus it is easy to expand a system with new ones or update the old ones.

To find the incapability issues in the design phase could reduce the project cost and the design time.

[2] http://www.cril.univ-artois.fr/en/software/abscon.en.html
[3] http://www.choco-solver.org/
[4] https://www.info.ucl.ac.be/ pschaus/oscar.html

Chapter 6

Assume-Guarantee Approach

System verification plays a significant role in system engineering. The correct behavior of a system (especially safety-critical systems) necessitates the fulfillment of the requirements based on the functional suitability, safety, or reliability. The complex safety-critical systems usually consist of multiple reusable or third-party components. The early detection of errors in these systems could save a lot of financial resources, and working hours.

In practice, model-checking paradigms are in use to detect and prevent errors during the system design phase. Model-checking is a type of the model verification techniques. Engineers build abstract behavioral models to establish properties of the system. A model checking tool can verify the required properties of a system against the model. The interaction of multiple components in a component-based system makes the model-checking more difficult and time-consuming. The relation between the number of states in the system to the number of the components is exponential. This limitation problem is known as “sate-explosion” problem.

To increase the scalability of model checking NASA have developed techniques [16][23] that decompose the verification task. These techniques use assumptions about the environment to check if a property is held for a component.

6.1 Technique

Assume-guarantee technique upholds a ”divide-and-conquer” approach [16] that deduces global system properties by checking the components in isolation. If each component satisfies its property, then so does the entire system. This technique has long held promise for modular verification [16].

Originally, Assume-Guarantee Reasoning was made to support the stepwise development of concurrent processes. Over time, multiple assume-guarantee reasoning based approaches have developed to cover the development process from the design-level verification [23] to the testing [16]. The assume-guarantee solutions cover the whole system engineering process (V-model). The design-level assume-guarantee artifacts help as a guide during the implementation phase and provide more efficient reasoning at the implementation level.

6.2 Artifacts

Assumptions and properties came from dynamic requirements. They formulated with dynamic modeling methods to show their behavior. This chapter uses labeled transition systems representing the assume-guarantee artifacts (components, assumptions, properties).

6.2.1 Assumptions and Properties

Usually, developers have behavioral information about the interfaces of the components. The environment (other components in the system) of the component is capable of invoking operation sequences in the component's interface. The term "Universal Environment" stands for the environment in which the component operates. The environment of the components could restrict the behavior of the component. The universal environment should call any service that a component provides and provide or refuse any service that a component requires.

Assumptions define and restrict the behavior of the universal environment. Properties are the acceptable and required operation of the component. If A is an assumption and P is a guarantee property, then $A \rightarrow P$ is the term that defines if P works correctly in the environment restricted by A.

With the terms of testing, assume-guarantee artifacts are interpreted as follows: Assumptions are functions used on test sequences (pre-invariants) to determine if the operational conditions of the component under test are correct. Properties (post-invariants) are the test oracles to check whether the component could accomplish the required operation. If the test sequences and the test results were both correct (valid), the component could guarantee its property and integrates well into the system.

6.2.2 Define and Generate Assumptions

Finding the proper assumption is difficult to find and could be nontrivial. The assumption must be more abstract than the component under examination, but still reflect the component's behavior. At the same time, the assumption must be strong enough for the other components to satisfy the property.

Techniques were created to generate the weakest environment assumptions [22] that enable the property to hold. Automatic assumption generation provides automated support for assume-guarantee reasoning. Assumption A_w holds for the weakest assumption [39] and characterizes all possible environments E under the assumption holds.

The generated assumption is safe (it should exclude all problematic interactions) and permissive (it should include all good interactions). The assumptions can be used [22] later during the run-time verification process.

6.2.3 Reusability

6.3 Assume-Guarantee Reasoning

The assume-guarantee formula is a triple $\langle A \rangle\, M\, \langle P \rangle$, where M is a component, P is a property and A is an assumption about M's environment. The formula is acceptable (*true*) if M is part of a system satisfying A, then the system must also guarantee P.

The simplest assume-guarantee proof rule [23] shows (**Proof 1**) that if $\langle A \rangle\, M_1\, \langle P \rangle$ and $\langle true \rangle\, M_2\, \langle P \rangle$ hold, then $\langle true \rangle\, M_1 || M_2\, \langle P \rangle$ is true.

The proof expressed as an influence rule:

1. Assume-Guarantee Reasoning.

$$\frac{\langle A \rangle\, M_1\, \langle P \rangle \qquad \langle true \rangle\, M_2\, \langle P \rangle}{\langle true \rangle\, M_1 || M_2\, \langle P \rangle}$$

.

Example[16] Figure 6.1 shows a communication channel that consists of two components (Left): M_1 and M_2. Property (Middle) P describes all legal executions of the channel in terms of actions. In this example: *in* must occur in the trace before any occurrence of *out*. The assumption (Middle) A comes from the design-level analysis of the system. Testing M_1 and M_2 in isolation shows that M_2 violates the assumption A in $\langle true \rangle\, M_2\, \langle A \rangle$ running the trace t_2 (Right).

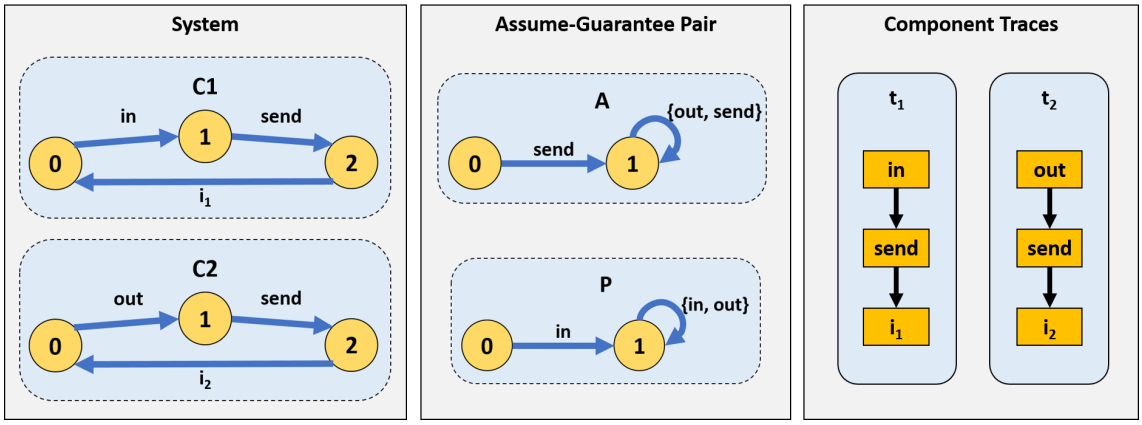

Figure 6.1: Communication Channel Example

Chapter 7

Design-Time Verification

7.1 Architecture and method

7.1.1 Component verification

The Assume-Guarantee Component Verification approach can check the system correctness through individual verification of the components. The architecture of this approach is shown in Figure 7.1. The next subsection presents the components of the architecture in details.

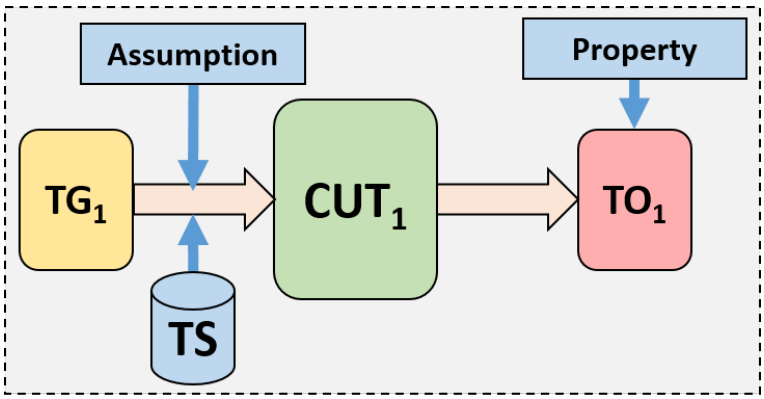

Figure 7.1: Component verification

Test Generator generates test paths reusing design-level assumptions and pre-defined Test Sequences. Without the Assumption, the assigned verdict of a test case could set to false positive or false negative because in this case, the context of the component is undefined.

Component Under Test refers to a component that is being tested for correct operation. The tested component usually an abstracted model of the intended part of the system.

Assumption describes the context in it the component designed to be used.

The pre-defined **Test Sequences** come from the design-level analysis of the component and its environment.

Test Oracle determines whether a component executed correctly for the generated test case. The Oracle defines two essential parts: Oracle information about the expected

output and oracle procedure that compares the Oracle information with the component property.

Properties are the requirements that the system design demands to guarantee correct operation.

This approach can be realized using finite state machines. All the components (Test Generator, Component Under Test, Test Oracle) modeled as finite state machines in this method.

7.1.2 Component Integration

In a complex software system, different components with different properties are working together. It is essential to verify that the properties of the integrated components stay true in the context of the system. Assume-Guarantee verification (testing) can obtain results on all interleavings of two individual component traces by checking each against the assume-guarantee premise.

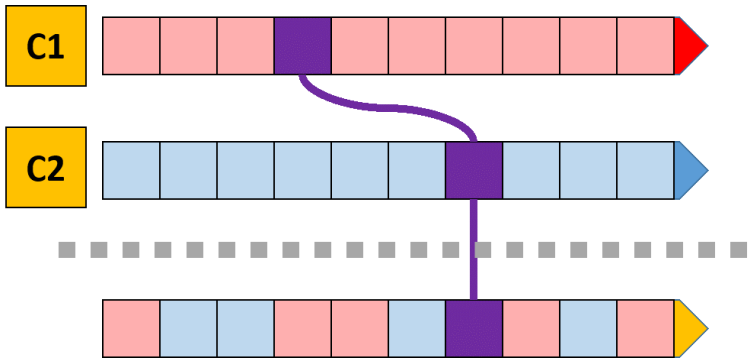

Figure 7.2: Trace Concatenation

Figure 7.2 shows two individual traces for the C1 and C2 components. The assume-guarantee based design-level analysis obtain these possible traces for each component. These traces could have common subsequences (purple). The interleaving of the traces could rely on the concatenation of the sequences joining the common subsequences.

Formally,

- $T_1 = \{x_1, \ldots, c_n, \ldots, x_N\}$ - set of test steps for component C1
- $T_2 = \{y_1, \ldots, c_n, \ldots, y_M\}$ - set of test steps for component C2
- $T = \{x_1, y_1, \ldots, c_n, \ldots, x_N, y_M\}$ - set of the concatenated test steps. The order of the elements are interchangeable.

7.1.3 Completeness

The component integration is complete if the following conditions are hold:

a) Test sequence corresponds to the assumptions;

b) Test results from different refinement level models correspond to the properties.

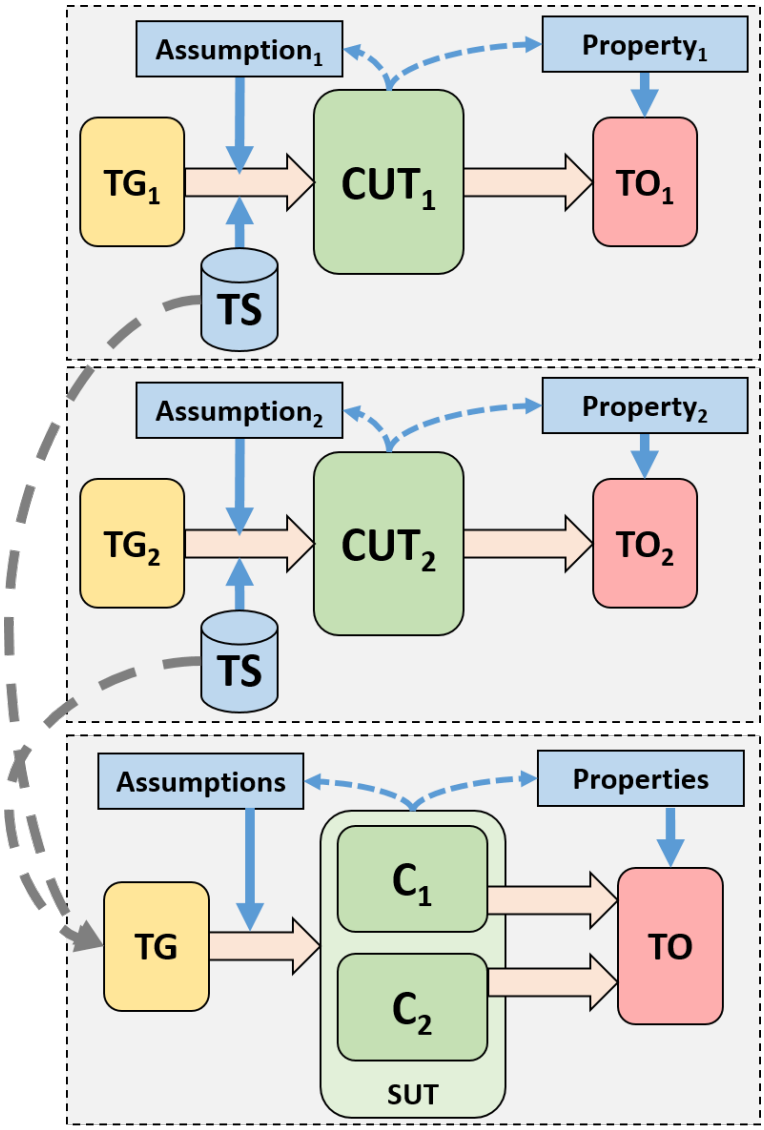

Figure 7.3: Component Integration

7.2 Verification and Testing in System Integration

Beside the design verification, component integration testing is also necessary to fulfill the dependability requirement of cyber-physical systems. Testing verifies the correctness of the implementation.

Assume-guarantee testing uses assume-guarantee reasoning to obtain system coverage and detect system-level errors. This testing method applies assume-guarantee reasoning to component test traces and uses the assumptions to generate the test traces for individual components. With this approach, system traces can satisfy a global property by checking the component traces in isolation. The restriction of the environment by using the assumptions restrict the false positive results. The violation of the assume-guarantee test result indicates system-level violations. Predictive testing project the system-traces onto component-traces to detect alternative trace interleaving without explicitly exploring them.

Method

1. Implement of universal environments
   (a) Implement U_A for component C_1, where the universal environment of C_1 restricted by the assumption A.
   (b) Implement U general universal environment for C_2
2. Execute the components in the universal environments to produce test traces.
   (a) Execute C_1 in U_A to produce the test trace set T_1
   (b) Execute C_2 in U to produce the test trace set T_2
3. Perform assume-guarantee reasoning
   (a) Checking each trace $t_1 \in T_1$ against property P
   (b) Checking each trace $t_2 \in T_2$ against assumption A

The method implies that $[t_1] \parallel [t_2] \models P$, for all $t_1 \in T_1$ and $t_2 \in T_2$

Chapter 8

Runtime Verification

8.1 Architectural Change Management

A lot of CPS component is dynamic reconfigurable during runtime so there is no possibility to check the integration during the design time. Furthermore, there is no time to check all possibilities during the design-time because of the very close deadline.

Several use cases necessitate the dynamic reconfigurability of a CPS. During the long service life of the CPS, several problems could occur. A fault, scheduled maintenance or a hardware update could involve the replacement of components that could necessitate the reconfiguration of the system. The need for dynamic reconfiguration could occur when the load or other external properties of the system are changing.

If a component has changed (reconfigured, new component), then it is necessary to update the related artifacts (assumptions, properties) to perform the verification.

The dynamic CPS middlewares (e.g., DDS, OPC UA) provides reconfiguration and automatic discovery. With the standard interfaces, they provide a good solution to resolve this problem.

8.2 Comparison with Traditional Methods

Model checking is a technique that suitable for showing that a system satisfies its specification. Model checking is a complicated task and in case of a large system often infeasible. Testing scales well to complex systems, however, it is impossible to cover all possible states of a system, because the system could have uncovered errors after the test phase.

Runtime verification is a combination of model checking and traditional testing. This technique allows checking whether an execution of a system satisfies or violates the given (correctness) property.

Runtime verification focuses on the detection of errors in the running system by creating monitors based on the correctness properties.

A correctness property is typically automatically translated into a monitor. Monitors are used to checking the execution of a system. Correctness property defines all observable executions of a system. Properties often expressed using formal logics, especially linear temporal logic which is well known from model checking.

Comparison of Model Checking and Runtime Verification Although model checking and runtime verification have a lot in common, there are many differences.

- While model checking corresponds to the language inclusion problem, runtime verification corresponds to the word problem. The inclusion problem has a higher complexity than the word problem because model checking should examine all executions of a system.
- While model checking deals with theoretically infinitely long execution traces, runtime verification investigates one running period (finite trace) of a real system.
- Runtime verification is unable to check the overall correctness of the system. However, detecting an error during the runtime verification indicates that the system operates incorrectly. Runtime verification only deals with observable traces.
- Runtime verifier has to obtain runtime information from the running system because it cannot directly evaluate atomic propositions in specific states. It collects all the information from interfaces
- While it is possible to execute the model checking in an early stage of the development, runtime verifications need to have the implementation.

Comparison of Testing and Runtime Verification Testing and runtime verification are both incomplete. They cover only a subset of the possible executions of the system instead of covering each possible trace.

In testing, a test suit has finite input-output sequences. Executing a test case checks if the output of the execution is equivalent to the predicted output.

Runtime verification shares more similarities with oracle-based testing. In oracle-based testing, the test suit has only input sequences. The designed test oracle observes the system under test and checks if properties are correct. The test oracle acts as a monitor in runtime verification. The biggest difference between the two paradigms is in the source of the oracle. While in oracle-based testing, the oracle is defined directly, runtime verification deals with generated artifacts from a high-level specification.

8.3 Assume-Guarantee Runtime Verification

Assume-Guarantee approach is capable of using in runtime-verification. The next section presents the method and the architecture (Figure 8.1) of the approach and shows a possible implementation. The method reuses properties and assumptions from the high-level design phase combined with the behavioral model of the components.

In the pilot implementation, statecharts were used to represent the runtime monitors and the behavioral model of the system. Gamma Statechart Composition Framework with DDS extension helps to generate the system implementation code and the monitor for the components from Yakindu Statecharts.

Each examined component has its properties and assumptions, so it is possible to check if a component violates the required property or the operation of the environment not correspond to the assumption. Both malfunctions indicate that the system does not operate as required. The violation of the assumption shows that the environment (component(s) of

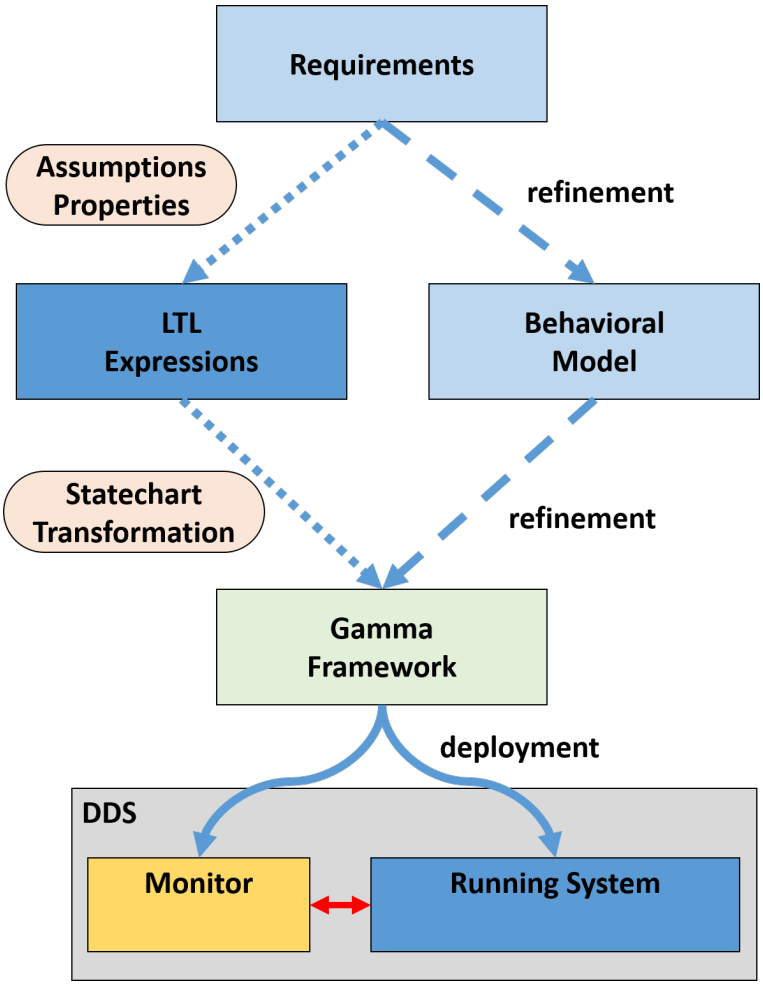

Figure 8.1: Assume-Guarantee Runtime Verification

the environment) cannot provide the proper operation for the examined component. However, the violation of the assumption refers to the malfunction of the examined component. The runtime monitor is originated in the assume-guarantee pairs.

8.3.1 Verification Method

This runtime verification method is based on the assume-guarantee approach. It uses the assumptions and properties to check if a component violates the required behavior. The assumptions and properties based requirements first translated to LTL expressions to provide a mathematical background to prove the correctness of the components. Later these LTL expressions translated to Yakindu Statecharts. The method uses the transformation process from [50].

8.3.1.1 Reusing the Assume-Guarantee Pairs

Assumptions and properties first defined at the early phase of the design process. They are used both in verification and testing phase. Assume-Guarantee verification and testing methods [16] [23] are using Labeled Transition Systems. It is possible to easily translate the LTSs to Yakindu Statecharts to integrate them into the runtime verification process.

LTS to Statechart Conversion Several papers were presented to transform statecharts to labeled transition systems to verify the behavior of reactive systems[18] [36][24][52]. They are dealing with complex, hierarchical statecharts. However, the complexity of the assume-guarantee based LTSs is low, so does the required statecharts. The translation between Labeled Transition Systems and Statechart requires to translate the LTS's states, actions, and transition relation into statechart specific blocks. Instead of extended states, these statecharts are dealing with simple states. LTS-based monitors are passive observers, so it is unnecessary to fire events or use guard conditions to allow or block a state transition.

One of the possible translation between the two system is the following:

- The states of the LTS directly translated into statechart states.
- The actions of the LTS are the action triggers for state transitions.
- The transition relation defines an unequivocal transition between states when triggering the corresponding action. Switching from one state to another in statecharts are using this relation when the action is triggered.
- The initial state of the LTS will be the initial state of the statechart.

8.4 Pilot Implementation

8.4.0.1 Statechart Composition

Gamma Statechart Composition Framework Gamma [5] is an integrated tool to support the design, verification, validation and code generation for component-based reactive systems. The behavior of each component is captured by a statechart while assembling the system from components is driven by a domain-specific composition language. Gamma automatically synthesizes executable Java code extending the output of existing statechart-based code generators with composition related parts.

Gamma offers the Gamma Composition Language (GCL) to describe components, interfaces and ports, and communication channels. Yakindu Statecharts translated to GCL with their respective ports and realized interfaces.

Gamma DDS Extension changes the communication interface to DDS. The deployed components are using DDS to interacting with each other, so it is possible to attach external components with DDS communication interface to the system.

Gamma Statechart Composition Framework was used to model the behavior of the components and model assume-guarantee pairs. Furthermore, it provides the opportunity to generate the code for the running system and the runtime monitor.

8.5 Pilot Example

The pilot implementation uses the Crossroad Example introduced in Section 1.2.1. The example is about to check the integration if the crossroad lights and the controller components. The method checks if the component violates its required property. This example shows how to use the concept with the Gamma Statechart Composition Framework. The workflow is the following:

1. Define the requirements of the Crossroad Example from the specification
   (a) Define the assumptions of the existing system
   (b) Define the properties for the new component (Pedestrian Light)
2. Transform the textual assumptions and properties into linear temporal logic expressions
3. Modeling in Gamma
   (a) Model the behavior of the components with the corresponding interfaces
   (b) Transform the assume-guarantee pairs from LTL expressions to Yakindu Statecharts
4. Generate the Java code for the monitor and system components
5. Run the system and the monitors

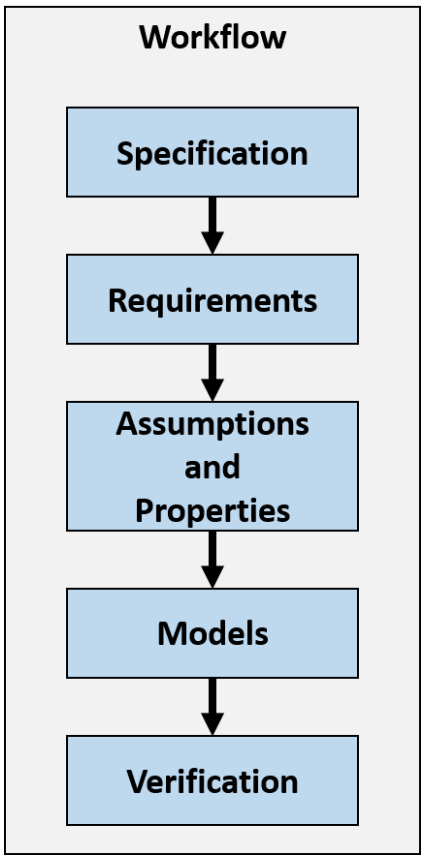

Figure 8.2: Pilot Workflow

8.5.1 Linear Temporal Logic Expressions

8.5.1.1 Properties

Properties describe the proper operation of the traffic lights. The traffics light operates forever in the following order:

$$\mathbf{gr}een \rightarrow \mathbf{ye}llow \rightarrow \mathbf{re}d \rightarrow \mathbf{gr}een$$

Color Exclusion Exactly one color is active at any given time:

$$\mathbf{G}(\neg(gr \wedge ye) \wedge \neg(ye \wedge re) \wedge \neg(re \wedge gr) \wedge (gr \vee ye \vee re))$$

Sequence The correct change of colors:

$$\mathbf{G}((gr\mathbf{U}ye) \vee (ye\mathbf{U}re) \vee (re\mathbf{U}gr))$$

Directions It is forbidden to get a green light in both directions (subscripts indicates the directions (P: primary, S: secondary)):

$$\mathbf{G}\neg(gr_P \wedge gr_S)$$

8.5.1.2 Assumption

The controller cannot toggle the lights while in a *Police Interrupt* state. The police officer should cancel the *Police Interrupt* state to continue the normal operation. Going back to normal operation requires two *policeInterruptToggle (piToggle)* signals. For the first signal, the system goes to the *Police Interrupt* state and the second signal triggers the normal operation

$$\mathbf{G}(piToggle\,\mathbf{U}\,piToggle\,\mathbf{U}\,(piToggle \vee liToggle))$$

8.5.2 Statecharts

The next step is to design and generate the Statechart models from the high-level requirements and LTL expressions. The safety (assumptions and properties) LTL Expressions translated to statecharts. In this example, I performed the model transformation by hand. However, it is possible to perform the transformation automatically with a parser.

8.5.2.1 System Components

Crossroad Light (Figure 8.3) iterates through the red-green-yellow-red sequence and switches to Interrupted mode when a Police interrupt occurs.

Controller (Figure 8.4) provides the synchronization of the traffic lights. It toggles the priority and secondary road lights with an offset.

8.5.2.2 Monitors

Figure 8.5 shows the assumption as a Yakindu Statechart. The created Statechart were extended with and *Error* state that indicates the violation of the assumption. The error state is expandable to raise an exception to control the operation of the system.

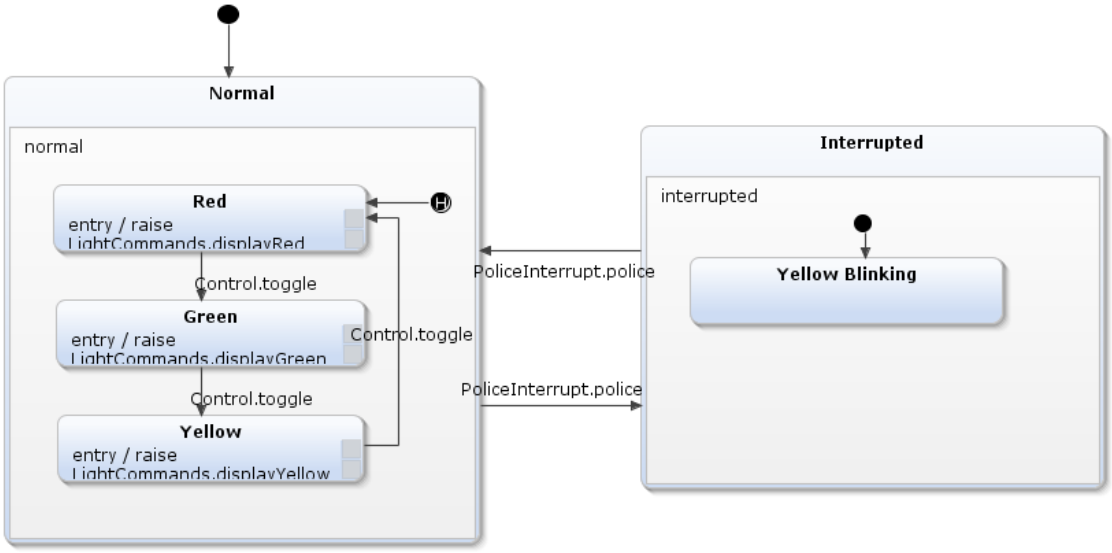

Figure 8.3: Crossroad Light

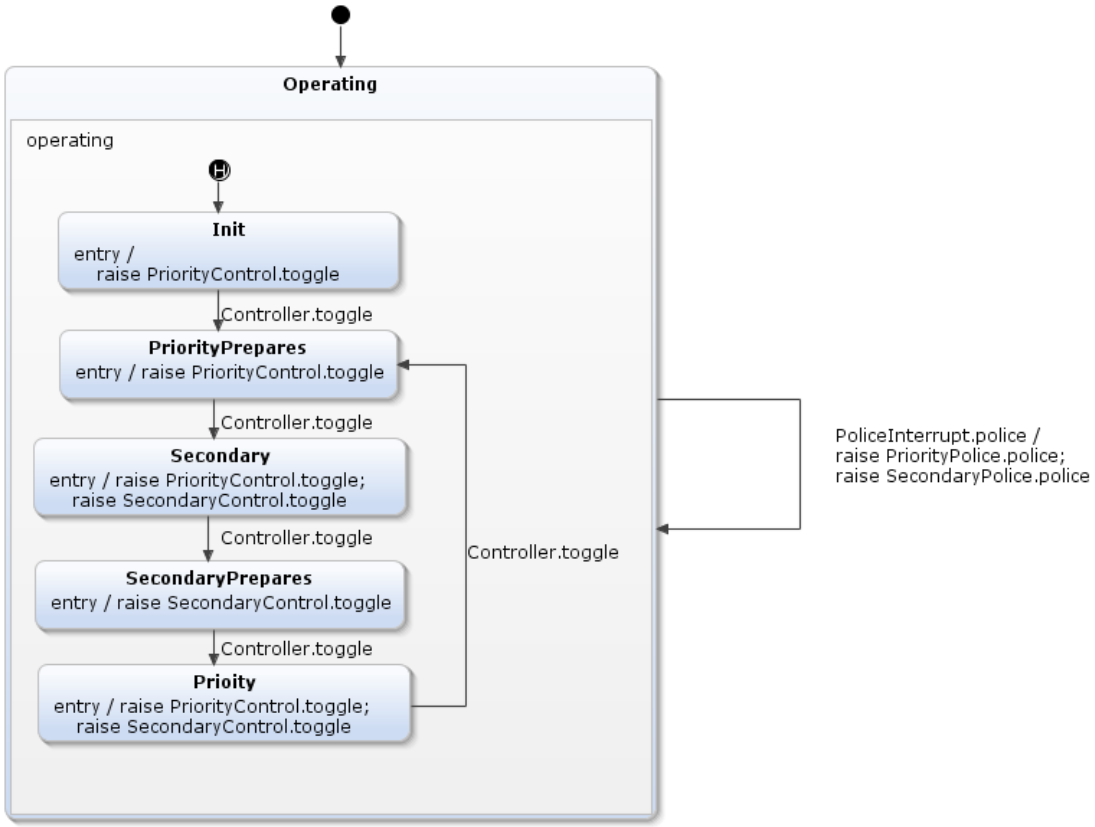

Figure 8.4: Controller

8.5.3 Evaluation

The implementation (statechart) of the controller contains an error. The error occurs in the following use case:

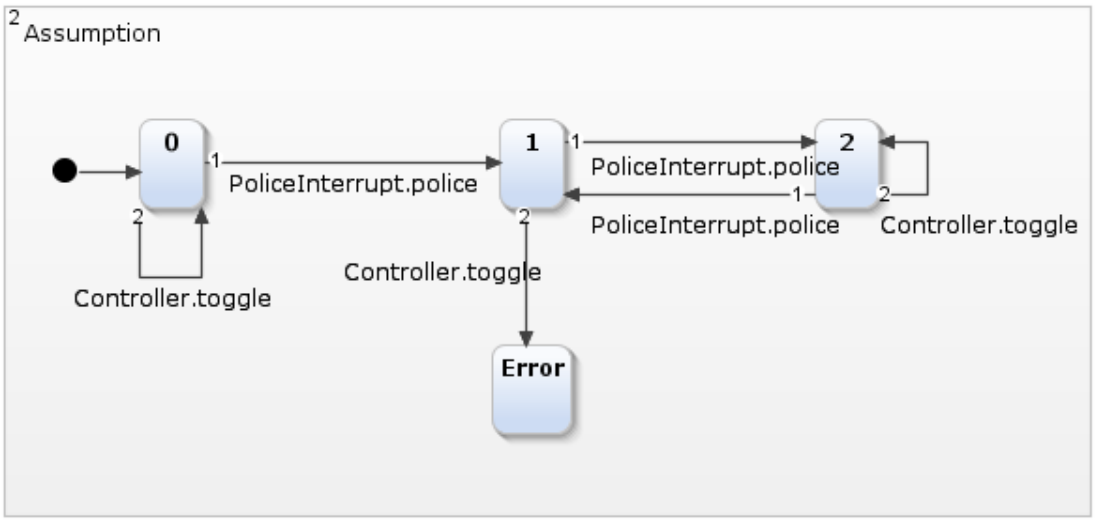

Figure 8.5: Assumption

1. The crossroad initializes, and the crossroad controller toggles the priority road to enter the Green state.
2. A police interrupt switches every light to a blinking yellow state.
3. The operator of the controller toggles the controller to next state (ignoring the police interrupt). The operator thinks the priority road has switched to yellow state, but it is still blinking due to the police interrupted.
4. The operator sends a toggle signal again. The operator supposes that the priority light is red and the secondary has switched to green.
5. Another police interrupt arrives. The traffic lights are returning into normal operation.
6. After two toggle signal both traffic lights switch to green because the traffic lights are not synchronized anymore.

The error originates in the Controller operation because it does not have a Police Interrupt state, so it is possible to send Controller toggle signals.

The assumption forbid the fire Controller Toggle signal after a Police Interrupt signal. The Assumption monitor state machines go to Error state indicating the malfunction of the system.

8.6 Integrating External Services

As cyber-physical systems are complex systems, integrating third-party components.

Due to the lack of information about the third-party components, it is not always possible to use the full assume-guarantee approach. It is possible to check the assumption to preventing the malfunction of the external service.

The next example presents how to integrate a black-box-like external service with the restricted assume-guarantee approach.

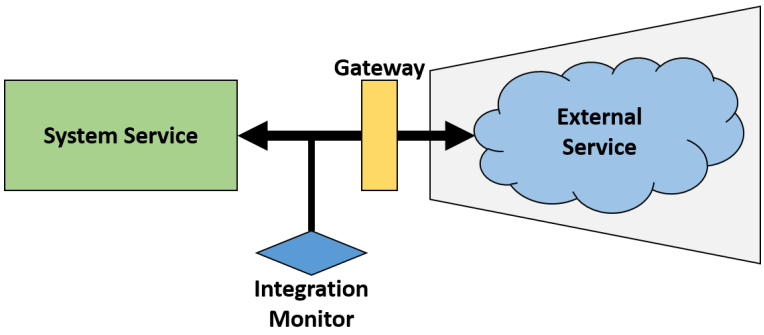

Figure 8.6: Integrating External Services

8.6.1 Image Classification Unit: Traffic Monitoring

In this example the Image Classification Unit uses neural network to implement the classification service. As the Image Classification Unit is a third-party component, it is possible to use the presented third-party integration method.

Neural networks can extract knowledge from a large amount of input and output examples, and then use it to process previously unseen inputs.

Training set is a set of examples (in this example images) used to fit the parameters of the examined model.

Usually, the neural network training set is a limited set of a complex domain (e.g., a car classification neural network uses only a limited number of car images for every class to build the network).

The neural network can serve with a miss-classification when its input is an out-of-domain image. It is possible to check the input data preventing this kind of operation.

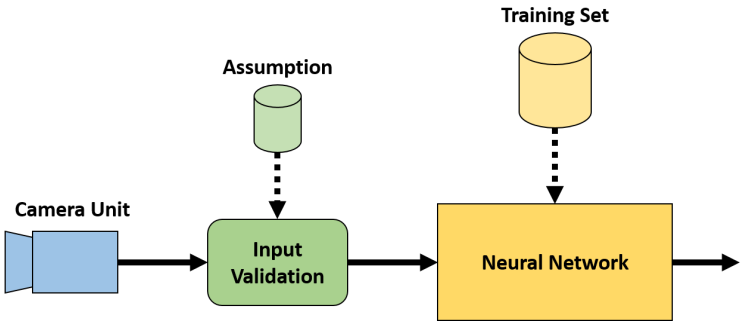

Figure 8.7: Classification Service - Architecture

It is possible to create assumptions about the input and decide if the input is the part of the domain or not. Restricting the input for the neural network can prevent a lot of miss-classification. Usually, light-weight image processing algorithms can be used to create assumptions.

Chapter 9

Summary

9.1 Further Research

The first step of the runtime verification is the identification of the errors (error detection). Extending the method with fault tolerance patterns makes it more effective and usable. [46] [45]

The example in Section 8.7 is an experimental example. The runtime framework is capable of integrating this type of problems, but the selection of the assumptions is not in the scope of this report.[27][51] The details of this ongoing research can be found in the scientific students' association report of Tamás Szántó.

9.2 State of the Work

Figure 9.1 shows the covered aspects by the presented verification methods. The green marker highlights the aspects that the dynamic methods cover (assume-guarantee based design-time and run-time verification). Furthermore, the yellow markers highlight the supported aspects by the configuration and static properties checking. The blue markers indicate the aspects covered by the presented platform technologies.

The following goals were accomplished:

- Exploration of model-driven and requirement-driven system engineering
- Using CPS Framework Middlewares (DDS, OPC UA)
- Elaboration of static and dynamic verification methods
- Elaboration of a runtime verification framework reusing assume-guarantee approach
- Implementation of the runtime verification framework rely on Yakindu based Gamma Statechart Composition Framework extended with DDS
- Present the methods with examples

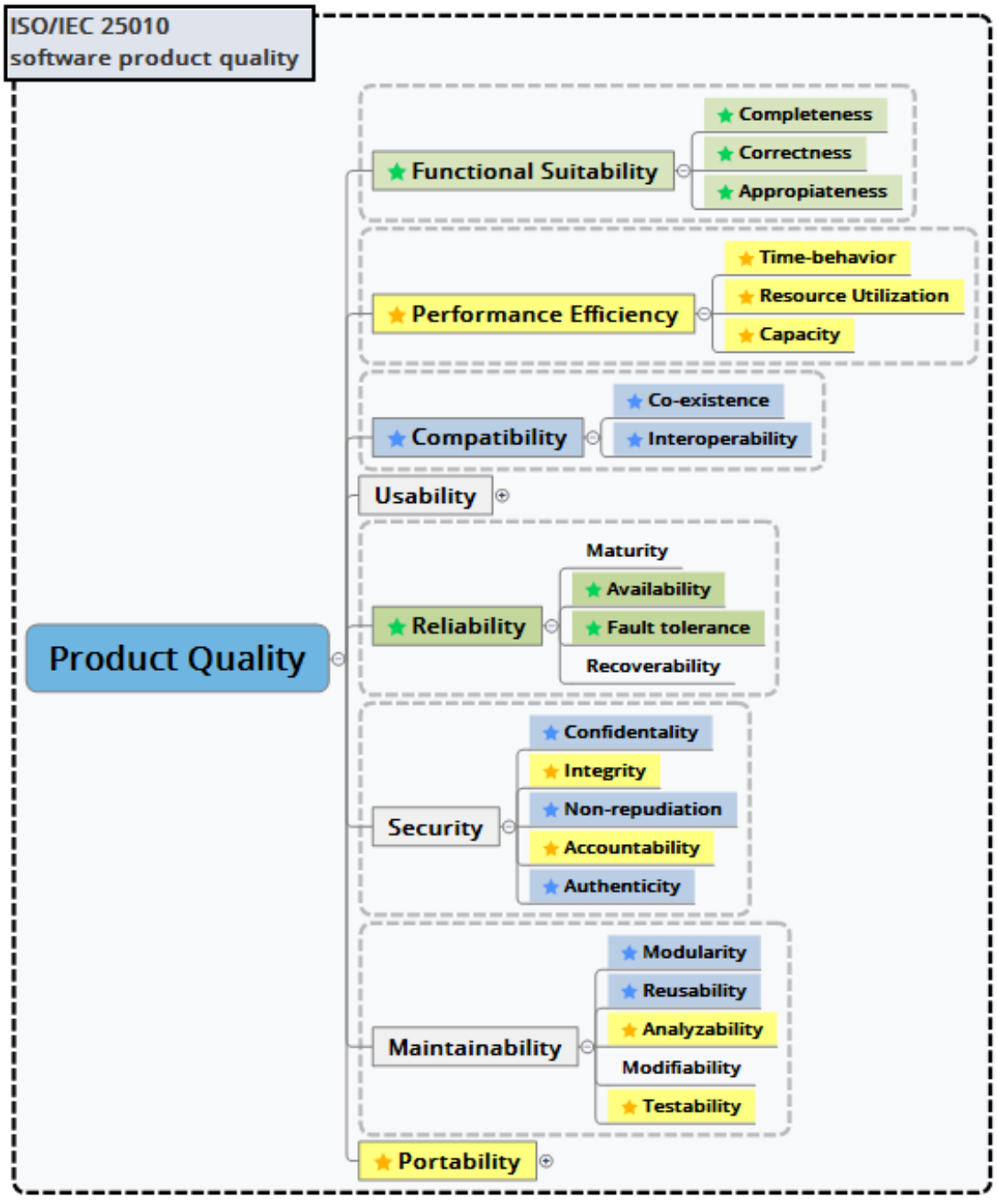

Figure 9.1: Evaluation of the Work

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Dr. András Pataricza for the valuable assistance and continuous support.

Furthermore, I would like to thank Dr. Josef Pichler (Software Competence Center Hagenberg - SCCH) for his great guidance and reviews, and the colleagues from SCCH for the help and the summer internship possibility.

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013) and the Pro Progressio foundation thanks to a donation of MAVIR Hungarian Independent Transmission Operator Company Ltd.

List of Figures

Bibliography

[1] CPS Architecture Layers. https://www.pubnub.com/wp-content/uploads/2017/06/edge-computing-diagram-1024x512.png.

[2] OMG Cloud Customer Architecture for IoT. https://www.omg.org/cloud/deliverables/cloud-customer-architecture-for-iot.htm.

[3] XCSP3 competition 2017. http://www.cril.univ-artois.fr/XCSP17/.

[4] OPC UA/DDS Gateway. https://www.rti.com/blog/announcing-the-opc-ua-dds-gateway-standard.

[5] Gamma statechart compostition framework. https://inf.mit.bme.hu/node/6028.

[6] IBM Rational DOORS. https://www.ibm.com/us-en/marketplace/requirements-management.

[7] Industrial Internet Reference Architecture. https://www.rti.com/blog/2017/01/31/2nd-version-of-the-industrial-internet-reference-architecture-is-out-with-layer

[8] MagicDraw. https://www.nomagic.com/products/magicdraw.

[9] OpenFog Consortium. https://www.openfogconsortium.org.

[10] OMG ReqIF. https://www.omg.org/spec/ReqIF/About-ReqIF/.

[11] ASQ/ANSI/ISO 19011:2011. Guidelines for auditing management systems. Standard, International Organization for Standardization, 2011.

[12] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[13] Fahiem Bacchus and Peter Van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *AAAI/IAAI*, pages 310–318, 1998.

[14] Santiago Balestrini-Robinson, Dane F Freeman, and Daniel C Browne. An object-oriented and executable sysml framework for rapid model development. *Procedia Computer Science*, 44:423–432, 2015.

[15] Andrew Banks and Rahul Gupta. MQTT version 3.1. 1. *OASIS standard*, 29, 2014.

[16] Colin Blundell, Dimitra Giannakopoulou, and Corina S Păsăreanu. Assume-guarantee testing. In *ACM SIGSOFT Software Engineering Notes*, volume 31, page 1. ACM, 2005.

[17] Andrea Bondavalli, Sara Bouchenak, and Hermann Kopetz. *Cyber-Physical Systems of Systems: Foundations–A Conceptual Model and Some Derivations: the AMADEOS Legacy*, volume 10099. Springer, 2016.

[18] et al. Bondavalli, Andrea. Dependability analysis in the early phases of uml-based system design. *Comput. Syst. Sci. Eng.*, 16(5):265–275, 2001.

[19] Ulrik Brandes, Markus Eiglsperger, Jürgen Lerner, and Christian Pich. *Graph markup language (GraphML)*. 2013.

[20] Rina Dechter and Avi Dechter. *Belief maintenance in dynamic constraint networks*. University of California, Computer Science Department, 1988.

[21] Michel dos Santos Soares and Jos LM Vrancken. Model-driven user requirements specification using SysML. *JSW*, 3(6):57–68, 2008.

[22] Dimitra Giannakopoulou, Corina S Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 3–12. IEEE, 2002.

[23] Dimitra Giannakopoulou, Corina S Pasareanu, and Jamieson M Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Proceedings of the 26th international conference on software engineering*, pages 211–220. IEEE Computer Society, 2004.

[24] Stefania Gnesi, Diego Latella, and Mieke Massink. Model checking UML statechart diagrams using jack. In *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on*, pages 46–55. IEEE, 1999.

[25] László Gönczy, István Majzik, Szilárd Bozóki, and András Pataricza. MDD-based design, configuration, and monitoring of resilient cyber-physical systems. *Trustworthy Cyber-Physical Systems Engineering*, pages 395–420, 2016.

[26] Edward Griffor, David Wollman, and Christopher Greer. Framework for cyber-physical systems. Technical report, National Institute of Standards and Technology - Cyber Physical Systems Public Working Group, 2016.

[27] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. A survey of methods for explaining black box models. *ACM Comput. Surv.*, 51(5):93:1–93:42, August 2018. ISSN 0360-0300. DOI: `10.1145/3236009`. URL `http://doi.acm.org/10.1145/3236009`.

[28] Akram Hakiri, Pascal Berthou, Andrew Gokhalec, Douglas C Schmidtc, and Thierry Gayraud. Supporting end-to-end scalability and real-time event dissemination in the omg data distribution service over wide area networks. *Journal of Systems and Software*, 86(10):2574–2593, 2013.

[29] James H Hill and Aniruddha Gokhale. Model-driven specification of component-based distributed real-time and embedded systems for verification of systemic qos properties. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[30] ISO/IEC 15026:1998. Information technology – System and software integrity levels. Standard, International Organization for Standardization, 1998.

[31] ISO/IEC 25010:2011. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. Standard, International Organization for Standardization, March 2011.

[32] ISO/IEC 26702:2007. Systems engineering – Application and management of the systems engineering process. Standard, International Organization for Standardization, 2007.

[33] ISO/IEC/IEEE 42010:2011. Systems and software engineering – Architecture description. Standard, International Organization for Standardization, 2011.

[34] Anish Karmarkar and Marcellus Buchheit. The Industrial Internet of Things Volume G8: Vocabulary. Technical report, Industrial Internet Consortium, 2017.

[35] Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, Peter Fritzson, Jörg Brauer, Christian Kleijn, Thierry Lecomte, Markus Pfeil, Ole Green, Stylianos Basagiannis, et al. Integrated tool chain for model-based design of cyber-physical systems: the into-cps project. In *Modelling, Analysis, and Control of Complex CPS (CPS Data), 2016 2nd International Workshop on*, pages 1–6. IEEE, 2016.

[36] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the spin model-checker. *Formal aspects of computing*, 11(6):637–664, 1999.

[37] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1): 99–118, 1977.

[38] Mark W Maier. Architecting principles for systems-of-systems. *Systems Engineering: The Journal of the International Council on Systems Engineering*, 1(4):267–284, 1998.

[39] Hoda Mehrpouyan, Dimitra Giannakopoulou, Guillaume Brat, Irem Y Tumer, and Chris Hoyle. Complex engineered systems design verification based on assume-guarantee reasoning. *Systems Engineering*, 19(6):461–476, 2016.

[40] Nitinder Mohan and Jussi Kangasharju. Edge-fog cloud: A distributed cloud for internet of things computations. In *Cloudification of the Internet of Things (CIoT)*, pages 1–6. IEEE, 2016.

[41] Roger Mohr and Thomas C Henderson. Arc and path consistency revisited. *Artificial intelligence*, 28(2):225–233, 1986.

[42] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The gamma statechart composition framework:: design, verification and code generation for component-based reactive systems. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 113–116. ACM, 2018.

[43] OMG SysML. OMG SysML. Standard, Object Management Group.

[44] Eila Ovaska, András Balogh, Sergio Campos, Adrian Noguero, András Pataricza, Kari Tiensyrjä, and Josetxo Vicedo. Model and quality driven embedded systems engineering. *Technical Research Centre of Finland*, 2009.

[45] Zsigmond Pap, Istvan Majzik, and András Pataricza. Checking general safety criteria on uml statecharts. In *International Conference on Computer Safety, Reliability, and Security*, pages 46–55. Springer, 2001.

[46] Zsigmond Pap, István Majzik, András Pataricza, and András Szegi. Methods of checking general safety criteria in uml statechart specifications. *Reliability Engineering & System Safety*, 87(1):89–107, 2005.

[47] András Pataricza, László Gönczy, András Kövi, and Zoltán Szatmári. A methodology for standards-driven metamodel fusion. In *International Conference on Model and Data Engineering*, pages 270–277. Springer, 2011.

[48] Héctor Pérez and J Javier Gutiérrez. Modeling the qos parameters of DDS for event-driven real-time applications. *Journal of Systems and Software*, 104:126–140, 2015.

[49] Trevor Pering, Kathy Farrington, and Thorsten Dahm. Taming the iot: Operationalized testing to secure connected devices. *IEEE Computer*, 51(6):90–94, 2018. DOI: `10.1109/MC.2018.2701633`. URL `https://doi.org/10.1109/MC.2018.2701633`.

[50] Gergely Pintér and István Majzik. Runtime verification of statechart implementations. In *Architecting Dependable Systems III*, pages 148–172. Springer, 2005.

[51] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144. ACM, 2016.

[52] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55 (3):357–369, 2001.

[53] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[54] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics). *Software & Systems Modeling*, 2 (3):187–210, 2003.