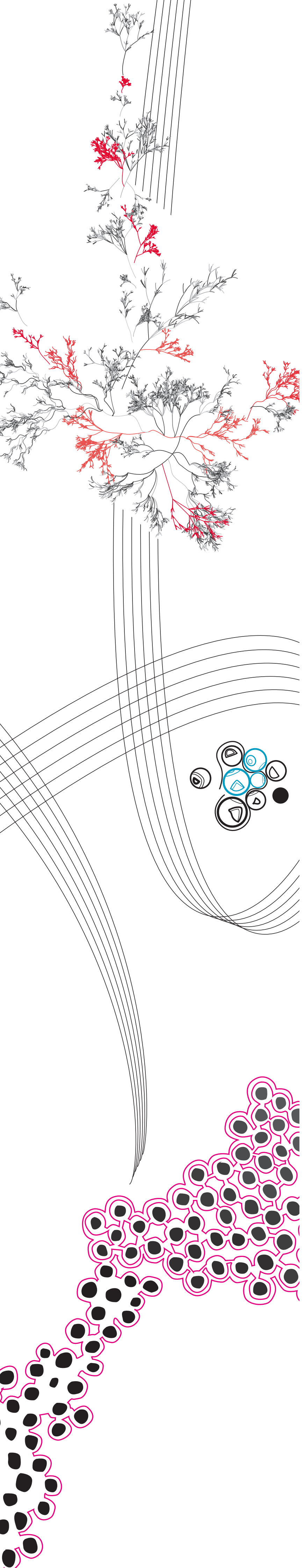


11-07-2018

MASTER THESIS

An Architectural Approach to Cyber-Physical System Design

BSc. T.J.W. Lankhorst <t.j.w.lankhorst@student.utwente.nl>
Supervised by
Dr. ir. B.J.F. van Beijnum (BSS)
Prof. dr. ir. G.J.M. Krijnen (RAM)

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Biomedical Signals and Systems

Exam committee:
Prof. dr. ir. G.J.M. Krijnen (RAM), supervisor
Dr. ir. B.J.F. van Beijnum (BSS), supervisor
Dr. ir. J.B.C. Engelen (RAM)
Ing. V.I. Sluiter (BME), independent member

Documentnumber
BSS — 17-05

UNIVERSITY OF TWENTE.

An Architectural Approach to Cyber-Physical System Design

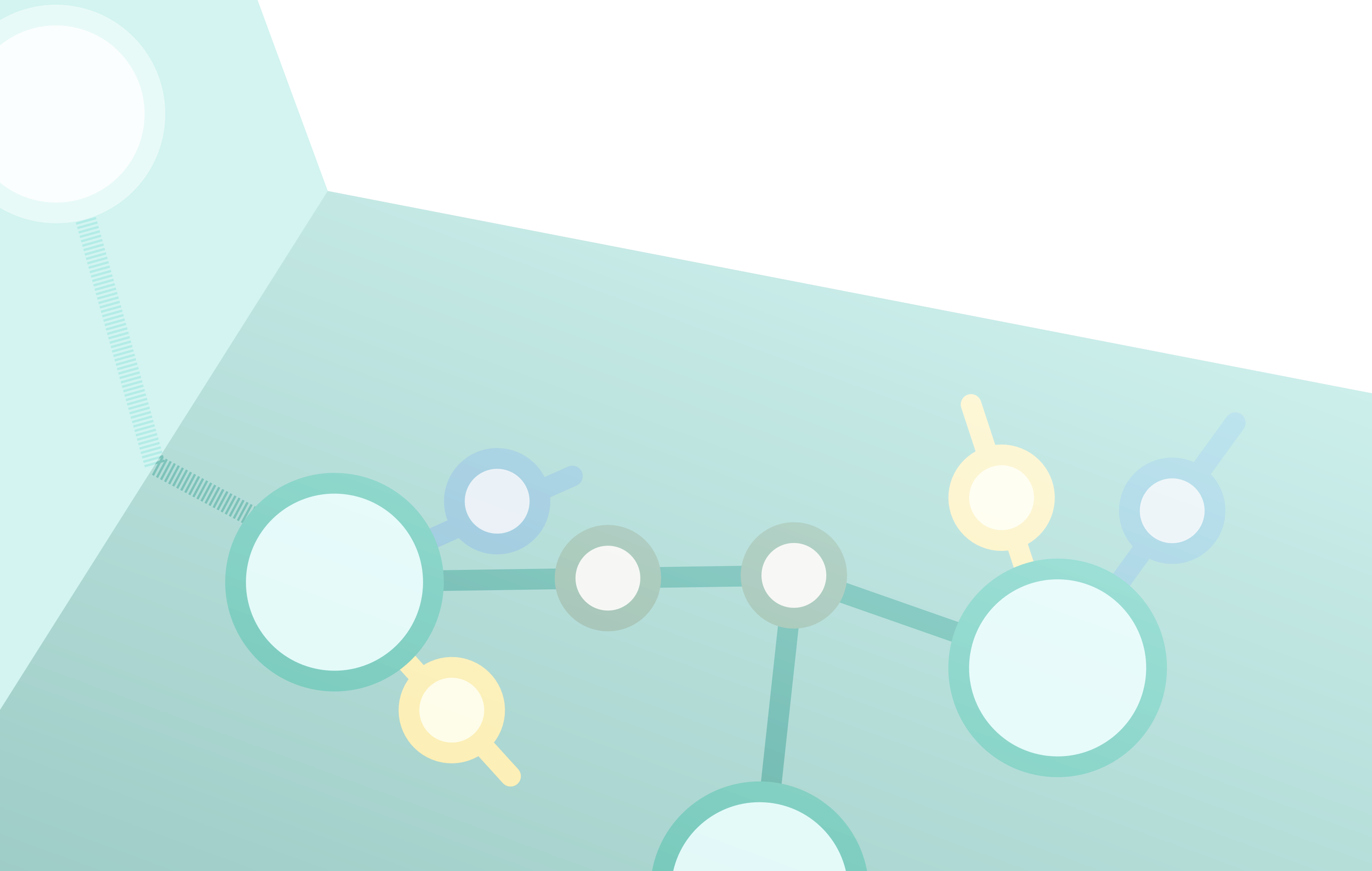
BSc. T.J.W. Lankhorst <t.j.w.lankhorst@student.utwente.nl>

Supervised by

Dr. ir. B.J.F. van Beijnum (BSS)

Prof. dr. ir. G.J.M. Krijnen (RAM)

11th July 2018



Abstract

Cyber-Physical Systems (CPSs) are systems that integrate distributed computation and communication with physical processes. They consist of networked devices that process data between sensors and actuators. They interact with their physical environment in an orchestrated, distributed and useful way. CPSs provide us with new possibilities that augment our world in ways that were previously unreachable.

Researchers and policymakers predict that CPSs will play a key role in solving social and economic challenges. These expectations attract the interest of industry and government, which stimulates research and investments in CPS development, applications and related technologies. Subsequently, new technologies and processes become available that take away technological and financial obstacles of CPSs. For example, processors and sensors become ever smaller, cheaper, more powerful and more efficient. But while financial and physical limitations disappear, new ones come into play: the growing complexity of CPSs makes their realization increasingly difficult. Human developers, designers and architects are limited in their ability to manage this complexity.

The objective of this research is to improve the design and management of CPSs by making its complexity more manageable. To do this, an Architecture Framework for CPSs is created. This framework aims at supporting the process of designing, testing and maintaining during the life cycle of the system. The framework consists of a selection of stakeholders and concerns and five viewpoints: the scenario viewpoint, the logical viewpoint, the process viewpoint, the deployment viewpoint and the development viewpoint. Application guidelines for using the framework are presented as well.

To validate the framework, it is applied to three use cases to evaluate how it supports a selection of eight characteristic concerns: effectiveness, modularity, consistency, reusability, extensibility, testability, understandability and simplicity.

Contents

List of Tables

List of Figures

Abbreviations

AD	Architecture Description.
ADc	Architecture Decision.
ADR	Architecture Decision Record.
AE	Architecture Element.
AF	Architecture Framework.
AFIoT	Architecture Framework for the Internet of Things.
AR	Architecture Rationale.
BSC	Binary Symmetric Channel.
CAD	Computer-Aided Design.
CAFCR	Customer., Application, Functional, Conceptual, Realization.
CPS	Cyber-Physical System.
CSP	Communicating Sequential Processes.
CT	Continuous Time.
DIKW	Data, Information, Knowledge, Wisdom.
DOF	Degree of Freedom.
DT	Discrete Time.
FMI	Functional Mock-up Interface.
FMU	Functional Mock-up Unit.
FSM	Finite State Machine.
HIL	Hardware in the Loop.
HPR	Hand-pose Reconstruction.
IMU	Inertial Measurement Unit.
IoT	Internet of Things.
KPN	Kahn Process Network.
LQR	Linear-Quadratic Regulator.
LTI	Linear Time-Invariant.
MADR	Markdown Architectural Decision Record.
MDD	Model Driven Development.
MIMO	Multiple Input Multiple Output.
MoC	Model of Computation.
OS	Operating System.
PID	Proportional Integral Derivative.
PoC	Proof of Concept.
ROS	Robot Operating System.
RTOS	Real-Time Operating System.
SDF	Synchronous Data-Flow.
SDLC	Systems Development Life Cycle.
SI	Software Intensive.
SIL	Software in the Loop.
SISO	Single Input Single Output.
SS	State-Space.
SSP	System Structure Parameterization.
UAV	Unmanned Aerial Vehiicle.
UML	Unified Modeling Language.
WG42	IEEE Work-Group 42.

1 Introduction

CPSs are complex spatial and temporal distributed systems that tightly integrate physical and software components. Though the current and possible applications of these systems are promising; the complexity of their design and management is the bottleneck in their adoption. *The architectural approach to CPS design* encompasses an Architecture Framework (AF) to improve the process of CPSs design and management.

As an introduction to the problem; this chapter describes the notion of CPS and provides an analysis of its design difficulties. Then; based on the analysis; it describes how this work contributes to reducing the design difficulties and what the scope of this contribution is.

1.1 Cyber-Pysical Systems

CPSs integrate computation and communication (the cyber aspect) with physical processes [1]. They observe and control themselves and their environment in a distributed manner to achieve their goals. CPSs find applications in all aspects of society.

A typical CPS is a heterogeneous network of *computers* equipped with *sensors*; *actuators* and *networking elements* that intertwine physical and software components. These components exhibit multiple and distinct behavioural modalities. Consequently; a CPS is a heterogeneous *system of systems*. Proper operation and regulation of CPSs requires careful *co-design* of its subsystems [2].

Intertwining of physical and software components is achieved through the use of sensors; actuators and networking elements. Sensors translate physical variables to the computer domain while actuators translate computer signals to the physical domain. Networking elements transfer data in the computer domain over physical media; they allow for internal and external communication. Internal communication between the CPS's components allows distribution of information. External communication between the CPS and other systems allows for retrieving information and commands. These other systems include computer networks; like the internet; or humans; through user interfaces. The CPS processes and analyses the obtained information to operate and regulate itself and its environment. Ideally; it performs better than a subset of its components. The physical environment that a CPS intertwines with is of critical importance. People or valueable objects might damage when a CPS malfunctions. For instance; a crash of self-driving vehicles in a smart-city; as visualized in Figure 1.1; has serious consequences for

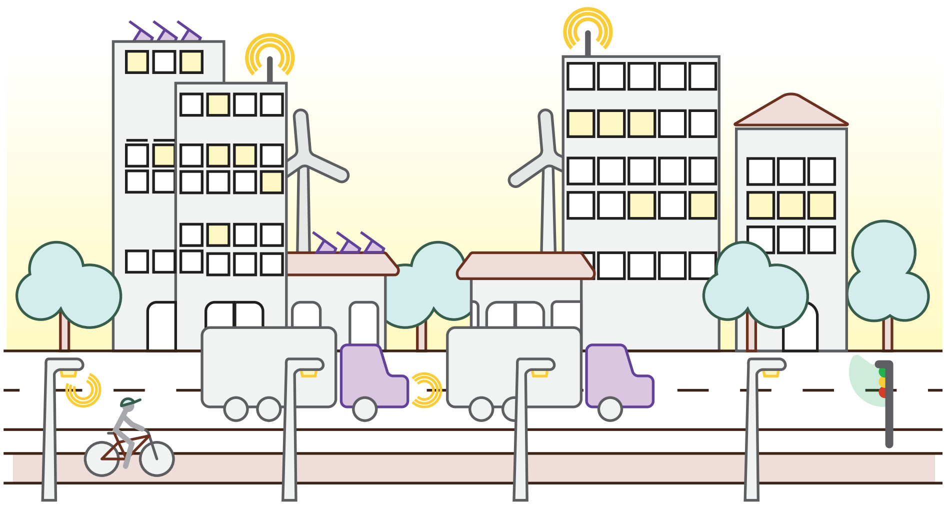


Figure 1.1 A smart-city is equipped with technologies that enable distributed data acquisition and control of all its aspects. Traffic-sensors; real-time energy-meters; litter detectors and presence detection are examples of sensing technologies that enable analysis and insights of the state of a city. These insights can then be used to anticipate and improve the usage of resources; such as energy and time.

the valuable resources that these vehicles contain. Because we want to avoid scenarios like these; a CPS must be dependable: trustworthy and reliable.

A CPS differentiates itself from systems with similar components in how it puts them to use. To make this more explicit; and to help differentiate CPSs from other systems; Torngren et al. [3] describes four descriptive aspects of CPSs:

- Technical emphasis
- Cross-cutting aspects
- Level of automation
- Life cycle integration

These aspects occur in any CPS; like in the smart-city described above:

Technical emphasis refers to the relevance of the technical aspects of the computation and communication components that a CPS consists of. Examples of such aspects are communication capabilities and optimization strategies. The potential of a CPS is in these technical aspects. Consequently; technical emphasis is a necessary aspect of CPSs. In a smart-city; communication and optimization strategies are not trivial. Handling unreliable communication from millions of sensors will require a specially developed infrastructure. Optimizing the resource usage of a complex system will require extensive modelling and analysis. The development and implementation of such communication and optimization techniques will require significant effort.

Cross-cutting aspects refer to the fact that CPS design involves a wide variety of disciplines such as control engineering; embedded software; sensors and actuators; networking and electronics. The involved engineers need to cope with the cross-cutting concerns rooted in these disciplines. Managing interdisciplinary dependencies between requirements is one of the main difficulties and main potentials of CPS [4, 5].

A smart-city is a perfect example of a system in which cross-cutting aspects are omnipresent. To optimize energy usage of a city; one must consider physical; electrotechnical; societal and business aspects.

Level of Automation refers to the *smartness* or *autonomy* of the system. In the vision of autonomic computing this is captured by the *self-star* properties of systems [6]: self-optimization; self-healing; self-configuration and self-protection.

Life cycle integration refers to the design approach that a system is not designed for its operational phase but also for its production; testing; maintenance and reusing phases.

In light of Torngren et al.'s four aspects; a smart-city is a CPS.

Further examples of CPSs include modern building automation [7]; smart agriculture [8]; Industry 4.0 [9] and the smart-grid [10]. CPSs can also be smaller scale like autonomous vehicles [11] and robotic prosthetics [12].

1.2 Difficulty of Designing and Managing CPSs

New and emerging applications of CPSs attract great academic and industrial interest due to their prospected economic; technical and social potential [13]. The evolution of CPSs is further stimulated by an increase in availability of hardware like sensors; actuators; computation resources and communication facilities. The production of sensors; for example; is growing exponentially: market forecasters expect a *trillion units per year* market within this decade [14]. At the same time; the availability of computational power is increasing as the cost; size and energy efficiency of processors and embedded devices improve. These developments will allow CPSs with more sensors; actuators and connected devices. Designing; implementing; validating and maintaining these systems will become increasingly difficult. Furthermore; the increasing number of connected devices presents us with new challenges like handling the resulting amounts of data; ensuring safety; security and resilience; coordination among groups of devices.

Lee [15] and Wolf [5] argue that in the long term; patching and improving the conventional computer abstractions (computer-centric approach) is not sufficient. Lee [15]; Derler, Lee and Vincentelli [16] and Rajhans et al. [17] agree on the need for stronger modelling semantics to adequately address the challenges of CPSs and cope with increasing complexity. Although academia and industry put much effort in CPS development research; like modelling and simulation techniques; formal methods for design; analysis and validation; software frameworks and middleware [2], this is not nearly a game played out.

1.3 A Framework to Design and Manage CPSs

In an effort to contribute to a solution for the problem of CPSs design and management difficulty; the remainder of this text concerns the following research goal:

Develop a framework to improve the design and management of CPSs.

To try to achieve that goal; the following research-questions will be answered in the form of an Architecture Framework (AF):

Q1 **What is a typical CPS**: what systems do we mean by CPS in this context and what are the common characteristics and problems of their design and management?

Q2 **What aspects *improve* the design and management of CPSs**: which aspects contribute to improvement of the design and management process of CPSs and how to quantify them?

Q3 **What are suitable viewpoints to cover these aspects**: how to look at the system to handle these aspects in a relatively independent way?

Q4 **Does the resulting AF indeed *improve* CPSs**: does application of the AF to use cases show that CPSs indeed benefit from the AF.

Today; companies and institutes use a mix of different systems-engineering methodologies and tailored frameworks to manage the development of CPSs. Standardized and specialized CPS management methodologies are still lacking as the industrial and academic interest in and adoption of these systems is relatively recent. Existing systems-engineering methodologies and frameworks; such as the V-model; Model Driven Development (MDD) and Agile methodologies; target towards more general systems and do not take into account the specificities of CPSs. Larger parties that involve in CPS design and management might modify and tailor these existing methodologies and framework or construct new ones but often keep the result in-house and proprietary.

In this report; we advocate *an architectural approach to cyber-physical system design*: a framework for managing CPSs throughout their life cycle targeted towards those that prefer to use a standardized methodology without having to fully tailor a methodology specifically to their needs. It encompasses an architecture based framework for creating and managing CPSs that provides guidelines for setting up a *work product* that motivates; documents; tests and creates CPSs: an Architecture Description (AD). Consequently; this document is an AF that aims at creating versatile ADs for CPSs. This AF encourages a way or creating and managing the AD of a CPS targeted towards individuals and small teams. It provides guidelines for using the resulting AD to reduce the complexity of creation and management during the CPSs' life cycle. The guidelines advocated in this document; including the tools used; are not binding; yet they serve as Proof of Concepts (PoCs) and starting point.

The central idea is that the AF addresses and simplifies problems that arise from CPSs' common characteristics. For example; CPSs involve communication and interaction with physical processes. These aspects bring forth problems that only partially concern the specific method of communication or the type of process. An effective AF addresses common issues while allowing specialized solutions to specific issues.

The inherent complexity and multi-disciplinary nature of a CPS makes it difficult to describe and analyse it as a whole. By describing the design problem from distinct perspectives; the AF simplifies problem description and analysis. The AF suggests five fundamental perspectives or viewpoints that are common to CPSs. These viewpoints constitute views that describe actual systems in an AD.

Architecture models are the components of a view. These models follow rules that an AF specifies or points out. An important goal of the AF is to impose model kinds that serve not only as building block for views; but as building blocks for the actual CPS too. This means that views will overlap with the actual system; up to the ideal situation in which an AD serves as system and vice versa.

1.4 Scope of the Framework

The development of this AF came forth out of a need for a structured way to work on:

Research projects that involve systems that have a cyber-physical character and in which structured architecture and information management is valued.

Such projects are about the development of new technologies and working principles that are tested using prototypes and elaborated in development and pre-market devices. The AF aims at improving these research-centric CPSs by providing a structured way of working; in the form of an AF; that guides the CPS design and development process; making the inherent complexity of CPSs more manageable and thereby allowing for a more efficient and effective development process that eventually leads to better systems. Evolution qualities - such as testability and extensibility - are assumed to be of essential importance to those systems that are within the scope of the framework. Chapter 3 describes the specific system aspects - in terms of stakeholders' concerns - that the AF supports to achieve its goals.

This AF is of interest to those that involve in the development and management of CPSs and that want to structure and standardize their development strategy without developing a tailored framework from scratch. The structure provided by an architectural approach typically benefits the quality of the system, its documentation and its rationale. In a project that is within the framework's scope, these qualities are valued.

1.5 Evaluating the Framework

The *successfulness* of the framework will be assessed by evaluating how well it improves the design and management of CPSs. To do this, the issue of 'design and management' will be represented by a number of concerns, being:

Effectiveness The framework should support development and analysis of the goals and functionality of the system.

Modularity The framework should support modularity to decrease the coupling and dependencies between subsystems and components. Modularity is associated with increased flexibility and reduced development effort [18, 19].

Consistency The framework should help in pointing out inconsistencies in the system as soon as possible to be better able to recover from them.

Reusability Well-designed components and subsystems must be reused to cope with increasing system complexity.

Extensibility The framework should provide measures to extend systems throughout their life cycle.

Testability The framework should provide means to make a system testable.

Understandability The framework should present the system in an understandable way.

Simplicity The framework should help in creating systems that achieve their purpose in a simple way. Simplicity is considered a prerequisite of reliability [20].

These concerns will be explained in section 3.2. Then, the framework will be applied to use cases that each involve a combination of these concerns. The use cases are then reviewed to determine whether and to what extend the framework tends to all concerns. Thus, the use cases will serve as an evaluation of the framework.

1.6 Architecture

This work explores advantages and disadvantages of putting architecture central to the creation of CPSs. The topic of *architecture* involves closely related concepts that may lead to confusion when not used precisely and consistently. To effectively discuss about the influence of architecture, this work adheres to a standardized description of architecture related terms and notions. The first section describes the meaning of architecture according to this standard.

Every system has an architecture. An architecture is the *conception of a system* that explains fundamental and unifying properties of the system. These properties explain why the individual components of a system form a whole and they describe the system's form, function and value.

Architecture is relevant to the system not only at the conception but throughout the whole life cycle of a system. A life cycle consists of phases that make up the lifespan of a system from its earliest conception until its end of life. A development process that takes into account distinct phases of a system's life is called a Systems Development Life Cycle (SDLC).

A wide range of life cycle models exists of which Thoben et al. [21] lists 7 in the context of CPSs. Of these, the ISO/IEC/IEEE 12207 standard [22] - focussing on software - and the Product Life Cycle Management for Internet of Things (IoT) [23] match the scope of this framework best. From these models, five phases are distilled which make up the life cycle model of a CPSs in this work (Fig. 1.2):

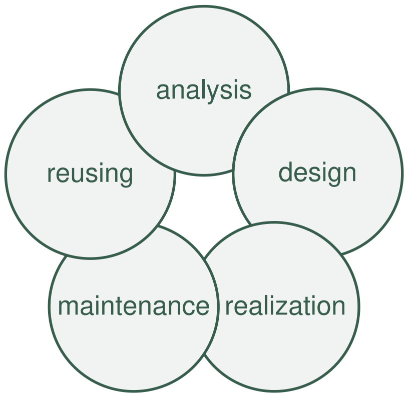


Figure 1.2 The life cycle consists of system engineering phases that focus on different aspects.

Analysis is about the process of determining the goals of the system of interest and what components the system needs to achieve these goals. Determining the goals typically involves the specification of requirements and expected behaviour which are the input to the design of tests for the system. This process includes the analysis of the feasibility of these components and adjusting the goals or components accordingly.

Design comprises the detailed description of the working of the systems' components. These components include the tests, specified at a higher level in the previous phase. The result of this phase consists of *building plans* of the components of the system and the system itself.

Realization is the phase in which the *building plans* are used to create the tangible components of the system that are realizations of the logical models developed in the design phase. Realization and execution of unit, feature and system tests should accompany the creation of the system's components. The result of this phase is a set of realized components integrated into a realized system with a corresponding test report.

Maintenance involves the use, preservation and repair of the system. Typically in this phase, the system is used to achieve the predefined goals. Tests find a use in this phase too: in checking the correct working during operation and after modifications or repairs. The output of this phase is a set of *results* that describe for example whether the system achieved the intended goal; the advantages and disadvantages of using the system and noticeable events and behaviour related to the system.

Reusing is the process of determining how the existing system, its components and its intellectual property influence the system and possible successors and correspondingly preparing the system for its intended reuse. Reuse may signify a combination of improving, adapting, archiving and deconstructing the system, its components and its intellectual property. When reuse involves significant improvements, adaptions or other modifications, it is followed up by an analysis phase again.

These phases form a cyclic pattern that emphasizes the *ever-developing cyclic nature* of most valuable and relevant systems. A specific system might profit from a refined or different life cycle model. The practises in this work that relate to one of the listed phases can be translated to a different model.

All but the most trivial systems need a plan to be successful. Developing and communicating such a plan quickly becomes impossible without a tangible (including digitally) work product. This *work product* is the Architecture Description (AD), attempting to express and convey the system's *conception*. Note that a work product is not limited to textual or graphical documents. For example, it could also concern a graphical model, physical machine or a combination of these. This is critical to allow for the ideal situation of the AD approximating the system (see 1.3). A saying that explains intuitively the relation between architecture and AD is: 'The map is not the territory' ([24]). We further describe ADs in the next section.

An Architecture Framework (AF) is a work product that conveys principles and practices for the description of ADs in a specific application domain. Another saying that explains the relation between architecture description and architecture framework is: ‘The legend is not the map’ ([24]). We describe AFs in section 1.6.1.

1.6.1 Architecture Description and Framework

The terms used conform to *ISO/IEC/IEEE 42010 - Architecture Description* [24], a standard developed by IEEE Work-Group 42 (WG42).

Architecture Description

An Architecture Description (AD) is a work-product that expresses an architecture. The form of this work-product depends on the context and may be whatever is helpful to the stakeholders. *Stakeholders* are entities that have interests in a system. *Concerns* explain the scope of these interests and describe their motivation. The *system* is the thing under consideration of an AD that *exists* either in hardware, software, logic or another form. It interacts with its *environment* which may consist of physical entities, software components, humans etc. An *architecture* captures a system’s concepts and properties in its environment. Stakeholders use ADs to help them understand and use the system of interest.

The ADs contains the following Architecture Descriptions (ADs):

Stakeholders are individuals or groups that have interest in a system through their typical set of concerns.

Concerns are interests in the system that are relevant to one or more stakeholders. These concerns include the purpose, feasibility, evolution, risks and impact of a system. They structure how the stakeholders define the success of a system.

Architecture Viewpoints are ways of looking at a system such that they frame concerns posed by one or more stakeholders. All stakeholders’ concerns have at least one framing viewpoint. The viewpoint consists of a set of model kinds. Viewpoints structure what aspects to show to treat one or more concerns.

Architecture Views are the result of looking at a system from specific viewpoints, consisting of models. Views face the system from a viewpoint such that it can be created, analysed or modified to comply to the respective concerns.

Model-kinds describes the goal, construction, rules and usage of models.

Architecture Models are initiations of model-kinds for a specific architecture that makes up the architecture views. Models yield the work product that describe the system and its architecture.

Architecture Rationale and Decisions list and explain key decisions in the AD. Rationale helps in understanding the architects their motivations behind these choices. Typical reasons to record a decision and its rationale include: the decision has architectural

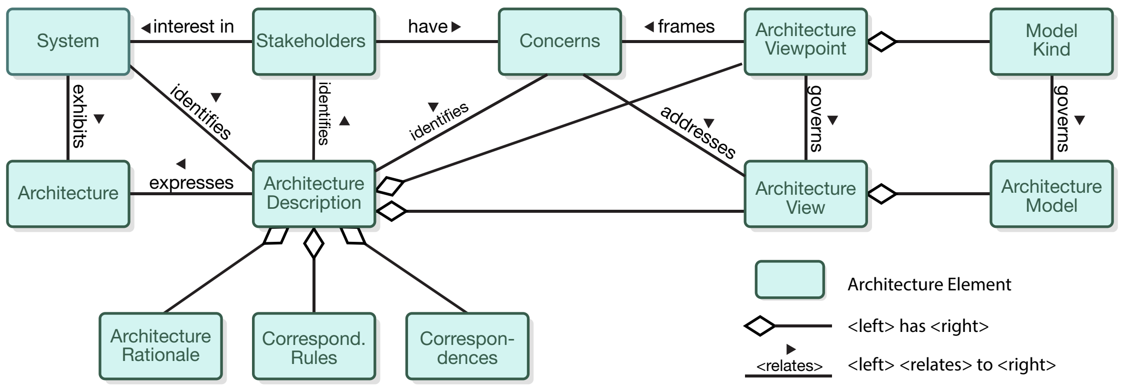


Figure 1.3 The structure of an Architecture Description, adapted from the IEEE 42010 standard.

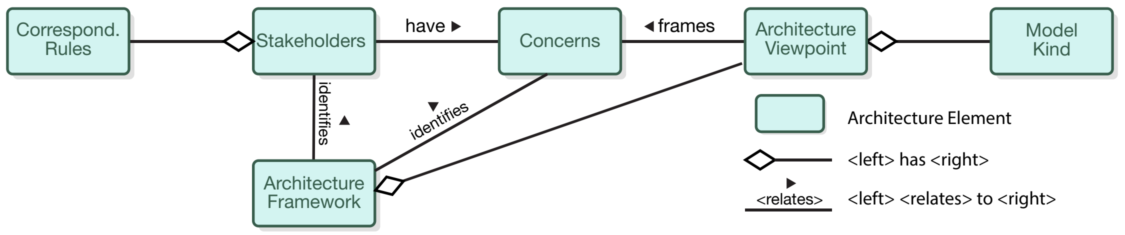


Figure 1.4 The structure of an Architecture Framework, adapted from the IEEE 42010 standard.

significant impact; the decision affects key stakeholders; the reasoning behind a stakeholder is exceptional or counter-intuitive; the decision has a high associated cost or risk.

Correspondence Rules are rules for architecture relations between AD elements (any element of an AD). Architects or design tools can check or enforce these rules to guarantee architecture consistency. Correspondence rules can also indicate design constraints of a system.

Correspondences are architecture specific instances of relations between AD elements.

Figure 1.3 shows the conceptual model of the architecture description.

Architecture Framework

An *Architecture Framework* describes conventions for creating and analyzing architecture descriptions. Its structure is comparable to that of the AD as shown in Figure 1.3 except that it does not identify a specific system nor express an architecture. As such, it does not include the architecture specific instantiations: views, models and correspondences. It also does not include design rationale.

Figure 1.4 shows the conceptual model of the architecture framework.

The 2010 revision of the standard expands on *architecture rationale and decisions*. Figure 1.5 shows the structural meta-model of the Architecture Decisions (ADcs) and Architecture Rationales (ARs).

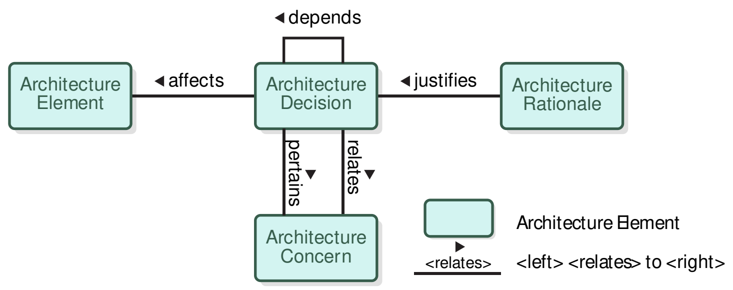


Figure 1.5 Meta-model of Architecture Rationale and Decisions, adapted from the IEEE 42010 standard.

1.6.2 Related Architecture Frameworks

WG42 collects examples of architecture frameworks. From this list, AFs that relate to the scope of this work are selected. After scoring out frameworks that target unrelated specific applications such as defense and enterprise architecture, three AFs remain: *IEEE P2413 âĂŞ Architecture Framework for the Internet of Things (AFIoT)*, KruchtenâĂŹs *4+1 view model* and *Customer Objectives, Application, Functional, Conceptual, Realization (CAFCR)*. The *5C* CPS AF [25] is another more recent but relatively well-known architecture.

AFIoT The Architecture Framework for the Internet of Things (AFIoT) is an AF developed by an IEEE standards working group that is in a very early phase of development. No official architecture documentation is released so there is no point in evaluating this AF.

4+1 View Model [26] is a framework for software intensive systems, shown in Figure 1.6. At the time of this framework's conception, the terminology was not standardized (see IEEE42010:2011), this is why the term *View Model* was used instead of AF. While the architecture framework targets software intensive systems, some of its design principles are applicable to CPS development as we will show in this paper. As its name suggests, the architectural model consists of 5 views. The *logical* view describes the classes, interfaces and collaborations that the system uses to achieve its goal. The *process* view describes the thread and process aspects such as synchronization and concurrency. The *development* view describes the organization of the software system in its development environment: libraries, components, executables. The *deployment* view describes how the software system maps to physical hardware. The fifth view *scenarios* is redundant with the others and connects the other views through use cases, scenarios and requirements. It describes the use cases of the system as seen by its end users, analysts and testers. These five architecture views express the design of the AD from different perspectives. The use cases in the scenarios view also help in validating the AD and act as guiding examples of applying the AD.

CAFCR architecture framework [27] builds upon five views: customer objectives, *what* does the *customer* want to achieve; application, *how* does the customer realize its goals; functional, the *what* of

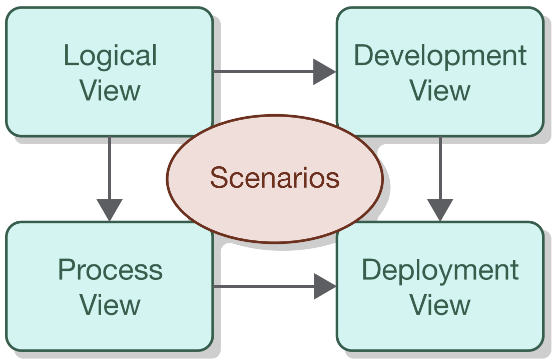


Figure 1.6 Kruchten's 4+1 view model

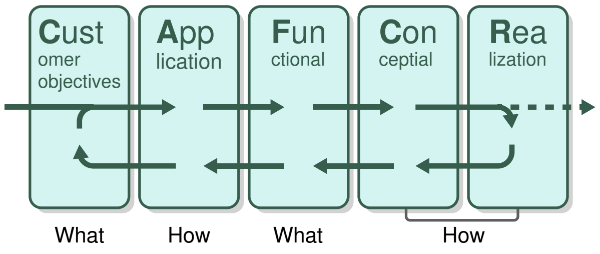


Figure 1.7 Iteration in the CAFCR framework. Viewpoints constitute a *what* or a *how*. The first two views are customer focussed, the latter three product focussed with the rightmost two viewpoints constituting a single *how*. The idea here is that a system's *how* is conceptually much more stable than the realization.

the product; conceptual and realization: the *how* of the product. From C to R, the views *drive* their following, from R to C, each view *support* their following. The *how* of the product is split in 2 views to increase stability: concepts change sporadically, realizations change often. Customer., Application, Functional, Conceptual, Realization (CAFCR) is focussed on a customer's world in a business context where the term *customer* describes any stakeholder. Figure 1.7 shows an iteration within the CAFCR set of viewpoints.

5C architecture [25] targets industrial CPSs and proposes a hierarchical set of five levels. The first level being *Smart Connection Level*, comprising sensors and actuators that form a sensor network of plug-and-play devices. Then, the *Data-to-Information Level* that involves analysis of data to assess component's health, wear, degradation and performance prediction. The *Cyber Level* combines the data from the previous layer with simulation models to discover anomalies and provide higher-level information. The *Cognition Level* translates the information and data to a human-interpretable format and allows collaborative diagnostics and decision making. Finally, the *Configuration Level* processes the information and control inputs to update the configuration of the system to amount for variations, changes and disturbances. The 5C architecture describes a number of layers and concepts but does not provide any tools that help in implementing an actual CPS.

Hilliard, involved in systems and software engineering and IEEE-42010 in particular, published a treatise on the *lessons from the unity of architecting* [28] in which he enumerates weak and strong aspects of architectural work products (frameworks, standards, methods, practices, life cycle models, systems).

A good architectural product is:

- *Minimal*: it assumes as little as possible, does one thing very well, is open and agnostic to combination with other products.
- *Precise*: it is clear about its scope, uses strict terminology and relations through an ontology (e.g. IEEE-42010 for AFs and ADs).
- *Contextual*: it takes all of its context, interests, stakeholders and concerns throughout the life cycle into account.
- *Separating Concerns*: the product facilitates the practice of system architecting by supporting *separation* of system *concerns* to make the system comprehensible.

Kruchten's 4+1 view model excels in its minimality, it provides support for separation of concerns. It is focussed on software-intensive systems and lacks context-awareness. The CAFCR framework excels in how it takes context into account, from stakeholder to realization. It features an interesting separation of concerns based on the stability of views. The 5C architecture provides a very clear CPS specific separation of concerns but does assume a industry-centric system context and lacks a precise use of ontology to clarify the framework.

Kruchten's 4+1 view model is elegant in its simplicity but not tailored towards CPSs. Yet, its choice of views is conceptually very interesting and might provide a starting point for a *minimal* AF for CPSs.

4+1 Viewpoints for CPSs

Kruchten's 4+1 view model targets Software Intensive (SI) systems. The previous revision of the ISO 42010 standard - IEEE1471-2000, targeted more specifically to Software Intensive (SI) systems - defined software intensive systems as *Systems in which software contributes essential influence to the design, construction, deployment and evolution of the system as a whole* [29]. According to this definition, a CPS has characteristics of a SI system. Software, however, is not the *only* essential influence to the development of a CPS. A CPS is about how both software, hardware and environment influence and balance each other. Its software focus is the main reason why Kruchten's 4+1 falls short for CPSs.

- The *logical* view describes the classes, interfaces and collaborations that the system uses to achieve its goals. In CPSs, distilling the required classes, interfaces and collaborations from the high-level goals of the system is non-trivial. Rather, the logical view should map concerns and use cases to *goals* and decompose these goals in lower-level goals and features, assisted by system requirements.
- The *process* view assumes the existence of threads and processes in an operating systems sense. A CPS is heterogeneous by definition and encompasses both the physical and the cyber world. The 4+1 view refers to processes in the software context of operating systems whereas in a CPS context, processes would refer to their

systems behavioural context, a unifying approach to both physical and cyber processes.

- The *development* view describes the components of the system used to assemble it. In typical software systems this involves executables, libraries and components. In CPSs, cyber-nodes host the software. Each cyber-node introduces specific software components such as sensor, actuator and communication drivers, middleware and application programs depending on the node type. A development view that serves CPS design should allow the designer to naturally specify the software structure for different types.
- In the *deployment* view the 4+1 model describes the hardware nodes at which the system runs. This view does not take the cyber-physical context of CPSs into account to express the relation between nodes, sensors, actuators, communication and physical components. In a CPS AF, the deployment view focusses on the structure (linking) of the system.
- The *scenarios* view describes the behaviour of the system as seen by the end user and tests through a set of use cases. The 4+1 view model suggests the use of Unified Modeling Language (UML) Use Case diagrams to denote this view. These diagrams provide the user with the possibility to specify the actions that occur between *actors* (parts of the system and possibly groups of users). Cyber-physical systems involve in complex interactions with the environment that a diagram with atomic interactions might not capture well. Other forms of diagrams are required as well. Examples are simulations and user stories.

The 4+1 view model specifies a usage *flow*: start with logical, then construct development and process, then construct deployment. In CPSs, the dependency between development and deployment is the other way around: the goals and the functionality that the system provides in cooperation with the physical world determines the deployment. The deployment in turn determines the development components.

Kruchten's 4+1 view model helped the development of SI systems but is not perfect for CPS design.

This work proposes a framework for CPSs that borrows from Kruchten's 4+1 view model its core separation of concerns, but tailored to CPSs in their full context and cast into the IEEE-42010 ontology framework.

Chapter 3 starts with a specification of stakeholders and concerns. Then, chapter 4 describe the framework's viewpoints and chapter 5 discusses application guidelines. Chapter 6 shows application examples for reference and validation.

1.6.3 Advantages of Standard Compliance

The AF that this document describes builds upon the foundations provided by the *ISO/IEC/IEEE 42010 Architecture Description* standard [24] that provides meta-models for ADs and AFs. Though the standard originally targeted software intensive systems, it now treats general systems engineering as well. The definitions and terminology used in this standard provide consensus and a starting point for this document. The AF claims compliance to the IEEE 42010 standard and aims at producing ADs that comply to the standard too. The standard specifies the exact requirements of a standard-compliant AD and AF. This includes the need to demonstrate whether the product satisfies these specified requirement when it claims compliance. Note that the requirements target contents, not form or organization. This is in-line with the goal of our *architectural approach*, to provide a structured architecture-centric way of working for CPSs that tightly integrates into their life cycle. Consequently, this document must evaluate whether the proposed AF satisfies the requirements. The requirements and their evaluation are listed in appendix A.

By complying to the IEEE 42010 standard, the AF aims at the following advantages:

- Its terminology and structure follow an accepted and standardized consensus.
- Its users can apply their knowledge of and experience with other IEEE 42010 compatible AFs and ADs.
- It can benefit from existing and future research, training and other resources that targets IEEE 42010 AFs and ADs.

1.7 Using Architecture to Design and Manage CPSs

The *Architectural Approach to Cyber-Physical System Design* is: using the Architecture Framework (AF) to create an Architecture Description (AD) for use during the life-cycle of the CPS.

Though the realization or translation of an AD to an actual system may bring problems of its own, it is assumed that they are either acceptable or trivially solvable: Any aspect of a system that poses non-acceptable or non-trivially solvable problems should be subject of the AD in the first place. If, for example, the realization of an AD is too costly then the issue of *cost* should be part of the AD.

This work focusses on the difficulty associated with conceiving the system and subsequently the system’s AD. The architectural approach aims to helps in addressing difficulties throughout the life cycle of a system. Benefits of the use of an architectural description include:

- It helps in specifying appropriate stakeholders, concerns and viewpoints for both the design of the system and the intended application of the system.
- It provides a systematic approach to multi-view design of systems. Relations between views help in assessing the impact of changes and the cause of problems.
- It provides a framework for simulation, synthesis and validation
- It serves as an information management framework, organizing the most important aspect of design: making decisions.

Successful use of the AF results in CPSs that perform better and are better manageable, in which the frameworks benefit outweigh the overhead of using it. Unsuccessful use of the framework means that the costs of its use do not outweigh its benefit, or worse, loss. Caution should be taken when selecting a methodology or framework: evaluate it, test it in small, discuss the selection with fellow stakeholders and reflect often. A framework is not recipe for success but rather a tool that is powerful when used right.

The purpose of this report is two-fold: first, it analyzes the problem of CPS complexity and proposes and evaluates a solution based on an AF; second, it acts as a work product that governs this AF. Appendix A recaps where the elements of the AF can be found in this document.

1.8 Structure of this report

Before we describe the AF we analyse the fundamental concepts of CPSs in chapter 2. In this chapter, the notion of a *general* CPS will be developed which will support the elements (Architecture Elements (AEs)) of the *architectural approach*. The following chapters describe the AEs of the framework and answer the research questions:

- Rationale of CPSs (chapter 2)
- Stakeholders and Concerns (chapter 3)
- Viewpoints (chapter 4, answers Q3)
- Guidelines for applying the AF (chapter 5)
- Use Cases, validation of the AF and application examples (chapter 6)

The rationale answers question Q1; stakeholders and concerns answers question Q2; viewpoints, consistency and correspondences answer question Q3. The use cases serve as application examples of the AFs and as an evaluation to check whether the framework indeed helps in managing the complexity of CPS development as to answer question Q4. In chapter 7 and 8 we provide respectively the discussion and the conclusion of this report.

This work contributes the following:

- Analysis of the foundational concepts of CPSs
- A new Architecture Framework (AF) for CPSs
- Best-practices for applying the AF
- Analysis of the AF throughout the CPS's life cycle
- Validation of the AF by application to three use cases

1.9 Summary

This section discussed CPSs and its design and management difficulties. It outlines an approach to manage these difficulties and explains the scope of this approach and the intended evaluation method. Then, the section discussed the concept of architecture and the work products Architecture Description (AD) and Architecture Framework (AF) according to the IEEE 42010 standard for Architecture Descriptions. Finally, it explained how an ADs can possibly benefit the design and management process and how this work serves as an AF for such ADs.

2 Rationale of A Framework for Cyber-Physical Systems

‘An effective Architecture Framework (AF) addresses common issues while allowing specialized solutions to specific issues.’ (section 1.3)

Determining the common *issues* that this framework needs to address, requires a specification of a general CPS and the involved *stakeholders*. This chapter reviews and connects a number of existing concepts that constitute a *base* for CPSs. The *base* CPS is an abstraction that represents the overlap between systems such that ideally, any CPS would extend this *base* CPS. A *base* CPS allows for statements about specific systems without knowing their full details. This *base* CPS serves as a premise for framework development. The *base* CPS is an abstract model of a real CPS thus, by definition, there is discrepancy between the simplified model and reality. This discrepancy might cause conflicts that, to be resolved, require knowledge about the working and the internals of the abstraction. In control theory for example, a controller might stabilize a linear model of a plant while the realization of this controller is not able to stabilize the real system. To resolve a conflict like this, the engineer needs to know about the assumptions of linear models, the implications of controller realization and uncertainties. So, by example it can be seen that models and abstractions are no substitution for knowledge about their working and underlying principles. When discrepancy between models, abstractions and reality affect a system, the discrepancies might be said to *leak* through: a phenomenon dubbed *the Law of Leaky Abstractions* [30]. This usefulness of abstractions and models is in that they help in getting more done with less effort as long as reality sufficiently complies to its abstraction. The abstraction that this chapter presents is a model of CPSs that is helpful but no substitution for in-depth knowledge as discrepancies between the model and actual CPSs will affect systems. In chapter 6 *use cases* and 7 *discussion* we will respectively encounter and discuss the consequences of these discrepancies.

Besides discussing the *base* CPS, this chapter provides supporting *rationale*.

Characterizing for a CPS is the *cyber world* This is the ‘space’ containing information about the system and allowing components to process and communicate this information. A cyber world mirrors the *physical world*, between these information flows. We discuss both the flow of information from physical to cyber and vice versa, then we discuss the groups of components of which a CPS consists. Finally, we discuss the inherent complexity and difficulties of this general notion of CPSs.

2.1 The Physical and the Cyber World

A CPS integrates computation and communication with physical processes. The physical components and environment of the system constitute the physical world. The system retrieves information from the physical world by combining sensor-readings with existing internal and external information. It uses the obtained information to track relevant aspects of the physical world through a model, by a simulation of the world. This simulation resides in the *space* made up by networked devices and memory: the cyber world [31].

Strictly speaking, the cyber world is part of the physical world too, for the processors, memories and communication buses eventually are physical devices. Being part of the physical world, other processes can influence the computation and communication of a CPS. Yet, the CPSs' cyber world is based upon computer abstractions that do not directly take the influence of these physical processes into account. These abstractions, called Model of Computations (MoCs), live at a higher level than that of electrical physical circuits because lowering their level of abstraction to such detail would impede modelling and analysis to an unworkable extend. The CPS' way to deal with the influence of physical processes on its computation and communication processes is to model them on the relevant process' level of abstraction. For example: electrical noise on a communication line can be modelled at the level of computer transactions as a stochastic process like a stochastic model that either successfully transmits our corrupts a message (such as a Binary Symmetric Channel (BSC)).

The cyber world is a model that mirrors the physical world. The concept of a world model saw early applications in robotics in the eighties: Kent and Albus [32] showed a world model that used sensor measurements and information in the system to make a model that tracks relevant aspects of the real world. The possibilities back then were limited, but today, through the increased availability of computation and communication, large-scale distributed world modelling becomes practical. System architectures based on cyber world models find application in industry and academics. An example of an architecture for CPSs that applied the cyber world model is Lee, Bagheri and Kao's *5C-architecture for CPS manufacturing systems*. This architecture bases upon the concept of cyber-twins that model their physical counterparts. These models are used to control, test, predict and understand the corresponding physical devices. Lee, Bagheri and Kao's architecture is discussed and compared in section 7.

This principle of interconnecting the physical world and the cyber world is illustrated in Figure 2.1. Note that the world model in the cyber world, being a model, does not have to imitate the representation our all of the detail of the physical world.

The increasing availability of computer devices, sensors and actuators combined with size and power reductions increases the granularity and distribution of the interface between the physical world and the cyber world. The interface becomes ubiquitous and invisible, it seems to fade. This is known as hybridization [33]. A CPS hybridizes the physical and the cyber world: it forms an interface between the two. To design a

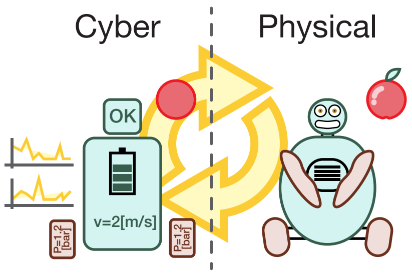


Figure 2.1 A CPS maintains a simulation of relevant aspects of the world through a world model constructed by combining sensor-readings with other information.

CPS architecture is to design an architecture for the interconnections between these worlds.

The cyber world is the total space of networked devices and memories that make up the CPS. It contains information that the system uses to achieve its design goals. The information in the cyber world can change over time. Devices provide new information to the world like sensor updates, failure states, goals and estimations. The cyber-world can span multiple connected devices and information distributes among them. Information does not necessarily have to be available to all devices in the CPS at all time. Such distributed systems require communication and coordination. Communication and coordination can significantly alter the behaviour of a system and therefore require attention. Communication between devices enables the exchange of information although not perfectly reliable. This is comparable to how a group of people in the physical world has distributed knowledge and how communication enables them to achieve goals that the members of the group can not achieve individually [34]. In a CPS, individual devices work together to achieve goals that they could not do individually.

After entering the cyber world through sensors and processors, the CPS processes and combines the information to model the world. The CPS uses the world model to help it in achieving its goals. These goals often include influencing the physical world through actuators. The system picks up these influences again through its sensors, closing the cycle. While the concept of a cyber world may sound at first instant relevant to only a small number of systems, it is actually common to almost any thinkable system that interacts with and controls parts of the world. To influence the world in a meaningful way, the system must know how to do this. May it be pre-programmed, open-loop our through an advanced control program, the system uses information about the world to influence it. This information is, in the case of a CPS, available in the memories of the networked devices known as the cyber world.

The model of a physical and cyber world found successful applications in both architectures and systems. It is simple yet it elegantly models general intelligent systems that tightly interact with physical processes.

2.2 Constructing the Cyber World

A CPS interacts between the physical and the cyber world. This section focusses on how sensing physical processes helps to construct the cyber world. The next section discusses the other way around: how the cyber world actuates physical processes.

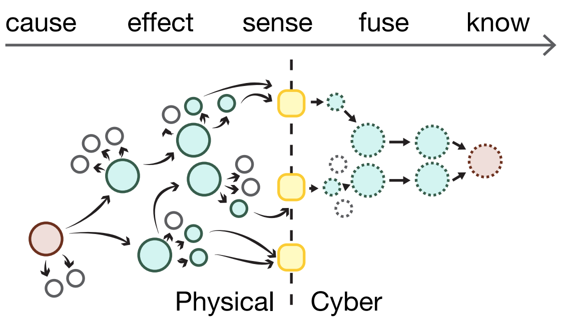


Figure 2.2 A physical event of interest causes reactions that sensors measure. The system fuses and combines these measurements to distil information about the particular physical event.

A CPS constructs its cyber world by distilling useful information from physical processes. This information consists of data about physical quantities and information about the world. Sensors take care of translating physical variables to computer-interpretable data. The CPS combines this with other data and information such as present prior knowledge and models to deduct further information.

Sensors enable the CPS to obtain low-level information by measuring physical variables like forces our temperatures. Though these physical variables might not be interesting in themselves, the events that cause them and their relations are. In general, low-level information produces higher-level information through combination and deduction. Computer vision, for example, uses two-dimensional matrices of intensity values known as images in sequences called videos to obtain information about the world at a certain time and place. In gesture recognition, we use time-series of accelerometers, gyroscopes and magnetometers to estimate intended gestures. Figure 2.2 shows how a high-level event of interest (such as a robot moving an end-effector) causes reactions (such as accelerations) that sensors measure. The CPS combines these measurements with each other and with information that is already available in the cyber space to deduct relevant information about their causes. Note that the implied causality of unidirectional influence of (sub-)systems in the physical part of the figure does not generally exist, instead the influence is bidirectional and simultaneous as we will see in section 2.5.

2.2.1 A hierarchy of information

Sensor and model information is combined to distil more useful information. The resulting higher level information might be used again to deduct conclusions. It is tempting to categorize the ‘level’ of information in distinct groups and include this hierarchy in the base CPS.

Indeed, researchers proposed different hierarchies to to distinguish discrete levels of information. A popular example is the Data, Information, Knowledge, Wisdom (DIKW) hierarchy [35]. Rowley [35] found the following common properties: data is symbols that represent observations; information derives from data and adds meaning to the perceived data; knowledge is a mixture of information, experience, skills and val-

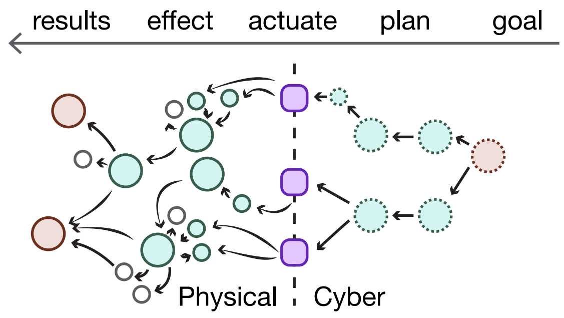


Figure 2.3 The systems needs to translate high-level goals to lower-level goals until the goals are so low-level that the system is able to achieve them by trivial control of its actuators.

ues. Frick [36] criticizes the hierarchy over its inconsistency in defining the different levels of the hierarchy.

The DIKW hierarchy suggests that there is a discrete number of levels; but this classification is highly application dependent. Furthermore, for some applications a different discrete classification our a continuous hierarchy is better suited. Because of these problems and application dependency, adopting the DIKW levelled hierarchy for the cyber world is not useful to a general CPS.

Note that although a single adopted classification scheme lacks, a CPS can adopt any scheme to its liking. Because such a hierarchy is very application dependent, it does not belong to a base CPS.

2.3 Controlling the Physical World

The goals of a CPS might require it to control our affect properties of the physical world. These goals are often formulated at a high-level such as "use the robotic end-effector to shake a user's hand" our "ensure that the load of the power-line network is optimally distributed". The CPS cannot achieve these goals by trivial control of actuators. It needs to combine high-level goals with information about the physical world. This information the system has available in cyber space and helps to decide how to control the actuators to achieve the system's goal. This approach is complementary to the way in which we modelled how a CPS constructs a world model in the cyber space: the CPS uses the cyber space world model to determine how it should control its actuators to achieve a particular result. Figure 2.3 shows this principle in a similar fashion as Figure 2.2. Again, note that the implied cause-effect relation of physical sub-systems of the figure does not generally hold. Rather, systems influence each other simultaneously.

In a typical robotic CPS, for example, a trajectory-planner translates the system's goal to end-point trajectories, then another module translates these endpoint trajectories to actuator trajectories and actuator control signals. To be able to control these different system levels, the responsible modules need access to system information like the kinematic configuration and interaction forces. This information is available in the cyber world. Constructing the cyber world and controlling the physical world are two interwoven processes.

2.4 The Complexity of the Physical World

We outlined a general model of how the CPS interacts with the physical world. The process of constructing the cyber world and controlling the physical world is difficult due to their complexity. We need to take into account this complexity when designing an architecture framework. Goldenfeld and Kadanoff [37] defined complex systems as systems that cannot be fully explained by understanding their components. The physical world is a complex system: understanding fundamental physics laws does not result in the full understanding of all aspects of the world. A CPS is a complex system as well: looking at each component individually does not result in fully understanding the system.

The CPS tries to extract useful information from the physical world. Goldenfeld and Kadanoff noted three modes of investigation to extract knowledge from a complex system: experiment, theory and computation. CPSs use all three modes to construct the world model.

Experiment Sensors instrument and actuators affect the physical world to find clues about the actual state of relevant aspects and the relation of the system with its (uncertain) environment.

Theory Theoretical models of relevant aspects of the world explain relations between measurements and observations.

Computation Simulation and estimation are computational methods that can help in understanding relevant aspects of the world.

These three modes make up the set of modes of investigation. In a CPS these modes complement each other. Classical examples of how combining experiment and theory help to investigate complex systems are Kalman filtering and outlier detection. These methods have clear applications in cyber-physical systems. Combining measurements and simulation is useful for comparison and anomaly detection. Theory and computation can help with validating each other. Choosing the right level of description is important in each of the modes of investigation. To be able to do this it must be clear what the actual goals are that the system tries to achieve and what questions it needs to answer: what are the system's concerns?

2.5 Dynamical Models

Models are useful for designers and for the system. A model can improve a designer's understanding, analysis and validation of systems while it can help a CPS with reasoning about measurements, goals and approaches. Models are most powerful when they are able to reflect complexity in system behaviour without becoming so complex that utility degrades [38].

The physical world is inherently complex and to model its aspects that are relevant to the CPS we need a systematic method. Willems [39] outlines an elegant hierarchical approach to systems modelling based on the three steps *tearing*, *zooming* and *linking*. The modeller tears a black-box system apart to identify smaller sub-systems and model them when appropriate. The modeller zooms onto each of these subsystems and recursively applies the *tear-zoom-link* sequence. Then, the modeller links the subsystems together. This approach is visualized in Figure 2.4.

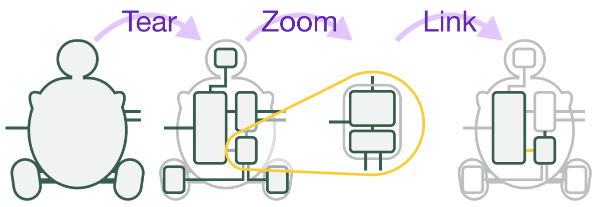


Figure 2.4 Tearing decomposes a model into sub-models, zooming is about modelling these sub-models and linking involves modelling the interconnections between these models.

Viewing the interaction of a system with its environment as an input/output relation is not an appropriate method to model dynamic systems. To show this we will look into models from a mathematical point of view.

A *mathematical model* is a pair $(\mathbb{U}, \mathcal{B})$ with $\mathbb{U}$ a set, the *universum* with elements called *outcomes* and $\mathcal{B} \subseteq \mathbb{U}$ called the *behaviour*. The behaviour is the restricted set of outcomes that are *possible*. If the universum follows from the context, the mathematical model can be taken as $\mathcal{B}$. A mathematical model is nothing more than a restriction of outcomes.

Consider a point in a two-dimensional space on a circle with radius r. A mathematical model for this constrained point's position is given by the universum $\mathbb{U} = \mathbb{R}^2$ and behaviour $\mathcal{B} = \{(x, y) \in \mathbb{U} | x^2 + y^2 = r^2\}$. The *represenation* of $\mathcal{B}$ is given by the formula $x^2 + y^2 = r^2$. The elements of $\mathbb{U}$ for which the equation holds belong to the behaviour. This is a *behavioural equation representation* of the mathematical model $(\mathbb{U}, \mathcal{B})$. In general, a behavioural equation representation of a mathematical model is denoted by $(\mathbb{U}, \mathbb{E}, f_1, f_2)$ in which $f_1, f_2 : \mathbb{U} \to \mathbb{E}$ and $\mathbb{E}$ is the *equating space*. In the point example f_1 is $x^2 + y^2$, f_2 is r^2 and $\mathbb{E}$ is $\mathbb{R}$. It is also possible for mathematical models to be expressed by behavioural inequalities our any other kind of description. Note that it is the behaviour, not the description of behaviour, that is the mathematical model.

A *dynamic system* is a mathematical model that relates time to outcomes. It is a triple

$$\Sigma = (\mathbb{T}, \mathbb{W}, \mathcal{B}) \tag{2.1}$$

where $\mathbb{T}$ is a subset of $\mathbb{R}$ called the *Time Axis*. $\mathbb{W}$ is the set of outcomes called the *Signal Space*. $\mathcal{B}$ is a subset of the universum, that is $\mathcal{B} \subseteq \mathbb{W}^{\mathbb{T}}$ where $\mathbb{W}^{\mathbb{T}}$ is the set of all maps from $\mathbb{T}$ to $\mathbb{W}$. Again, the behaviour is central. It formalizes which trajectories $w : \mathbb{T} \to \mathbb{W}$ agree with the model.

Some models need additional variables besides the *manifest* variables that the model aims to describe. These *latent* our internal variables can be included in an extended definition of a mathematical model: A *mathematical model with latent variables* is a triple

$$(\mathbb{U}, \mathbb{U}_l, \mathcal{B}_l) \tag{2.2}$$

with $\mathbb{U}_l$ the universum of latent variables and $\mathcal{B}_f \subseteq (\mathbb{U} \times \mathbb{U}_l)$ the full behaviour. The *manifest mathematical model* is $(\mathbb{U}, \mathcal{B})$ with the *manifest behaviour* (our behaviour) $\mathcal{B} := \{(u \in \mathbb{U} | \exists l \in \mathbb{U}_l) \text{ s.t.} (u, l) \in \mathcal{B}_f\}$.

Applying this notion to dynamic systems we obtain a definition comparable to (2.2). A *dynamic system with latent variables* is a 4-tuple

$$\Sigma_{\text{full}} = (\mathbb{T}, \mathbb{W}, \mathbb{L}, \mathcal{B}) \tag{2.3}$$

with $\mathbb{L}$ the set of latent variables and the full behaviour $\mathcal{B}_f \subseteq (\mathbb{W} \times \mathbb{L})^{\mathbb{T}}$ a subset of the set of maps from time to the Cartesian product of the sets of signal and latent variables. A special case of a dynamic system with latent variables is a state-space model. This model relates the variables of interest (manifest), input and output, through a set of hidden (latent) variables, the state.

Systems with latent variables are essential in modelling physical systems. Although latent variables might be possible to eliminate, this is not always helpful our possible. Latent variables can provide more insight into the system, as is the case with state-space models.

2.6 Interconnecting Dynamical Models

As described above, there are interactions going on between the physical and the cyber world. We have explained these interactions intuitively by input-output and cause-effect. This kind of explanation kept a central place in systems and control theory throughout the past century. Although mankind has been able to solve many problems using this approach, it does not describe the true nature of the behaviour of complex systems. Not signal transmission but variable sharing is the universal foundation of complex system interconnection [39]. Explaining a mathematical model by inputs and outputs is possible when an input/output partitioning exists. Finding such a partition is the approach taken in bond-graphs models: components share effort and flow variables which the modeller partitions into input and output such that the modeller can simulate this variable sharing by input-output relations.

A CPS constructs a world model. To sufficiently model physical phenomena, it has to take into account the *variable sharing* nature of these phenomena. Thus, components that model interconnected systems should support bidirectional communication when an input/output partitioning is not possible.

Let us describe the *tear-zoom-link* approach to interconnected systems modelling as introduced above. The main components are modules and terminals. A module is a model that specifies the behaviour of the variables that live on the terminals. The behaviour relates the variables on the terminals to each other. A terminal has a type such as *electrical*, *3D-mechanical* our *thermal*.

The *interconnection graph* describes the connection between models and terminals:

$$G = (V, E, L) \tag{2.4}$$

G is a graph with leaves. V is the set of vertices that represent modules, $E = \{\{x, y\} | x, y \in V\}$ is the set of edges that represent connections between terminals and $L = \{x \in V\}$ the set of leaves representing open connections of terminals.

E may contain self-loops ($e = (\{x, x\}, t)$), representing that a module has two connected terminals, which contribute 2 to the degree of the vertex x. The degree of vertex equals the number of terminals that a module has. Two terminals, of physical type, that are connected (represented by an edge) must have the same type. This graph provides us with a system of connected modules that specify their behaviour.

Connecting this system to other systems starts by defining the interconnection equations which follow naturally from the terminal type: in electrical terminals Kirchhoff's laws hold and in logical connections the input must equal the output.

The interconnection equations together with the module equations give rise to the behaviour of the connected system. We now specify the manifest variables as a function of the terminal variables. The terminal variables are thus considered latent. The module and interconnection equations and the manifest variable assignment define again the full behaviour of the system $\mathcal{B}_f$.

Now that we connected the modules together we are ready to zoom in on a module and decompose it, zoom out off the system to connect it our be happy with the result.

2.7 Signal-flow Models

Variable sharing is the correct way to interconnect dynamical models. The main reason why the input-output approach survives in engineering and research is that it can describe certain classes of systems sufficiently. These classes describe unilateral phenomena like transistors and amplifiers. If the omitted physical details in the simplification are not of significance to the problem, there is no problem in using such a simplification. Some systems and phenomena are better described by input/output models than by dynamical models. This is in line with Goldenfeld and Kadanoff's notice: choosing the right level of description is important. Like a biologist should not model the human body at the level of atomic interactions, a computer engineer should not model a computer at the level of electrical networks of transistors and other electrical components. Systems that have a typical direction of signal-flow are generally not modelled well by dynamical systems. These systems are better modelled at a higher level of abstraction like *discrete events*, *synchronous dataflow* our *finite state machines*.

Interconnecting systems with signal flow is a matter of connecting inputs to outputs and vice versa. To allow for signal flow terminals in the interconnection graph we, two terminals of opposite (input/output) type need connection through an edge $e \in E$.

The cyber-world of a CPS consists of computing and communication devices: devices that, at their core abstraction, have unilateral influence on each other. This might be one of the most important differences between the physical and the cyber world. The physical world is typically modelled by interconnected dynamical system models whereas the cyber world is modelled by interconnected signal-flow system models. Despite this essential difference, sub-systems in the cyber world rely heavily on analysis and simulation of physical models.

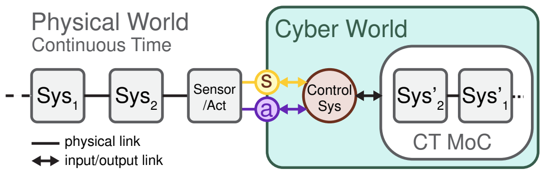


Figure 2.5 The cyber world and the physical world interact through sensors (s) and actuators (a). The cyber world might analyse and control the physical world through models. These models are encapsulated in the cyber world's own MoC.

The common situation of mixing different MoCs, as illustrated in Figure 2.5 emphasizes the need for a way make these abstractions and their connections explicit. Model of Computations (MoCs) play a central role in this.

2.8 Interconnecting Different Types of Models

We discussed the modelling and interconnection of dynamic systems and showed that the interconnection and behaviour of dynamic systems follow a set of rules. We showed that a different type of model, signal-flow, yields a different set of rules. In a heterogeneous system, like a CPS, we want to combine these different types of models. Be that as it may, interconnecting different types of models is non-trivial.

Take dynamical systems and signal-flow systems. The typical way to connect a signal flow output to an electrical circuit is to use a model of a modulated voltage our current source. The target effort our flow is set by computing the flow our effort. This is a design choice that may not reflect the physical possibilities and impossibilities of this interconnection. As soon as there occurs respectively a short-circuit our disconnection, the interconnection would yield infinite flow our infinite effort which is physically impossible. The designer has to take this into account when interconnecting different types of systems. Another example is connecting an electrical network to a signal flow input. Measuring an effort our a flow is modelled by a sensor that reads variables without affecting them. In practice, sensors do affect the dynamics of the dynamical system. The designer has to take care this effect is insignificant on the total system behaviour.

2.9 Models of Computation

The previous sections introduces two different types of models: dynamical models and signal-flow models. The semantics of interconnection and behaviour differ per type of model. MoCs provide a formal way of describing these semantics. A MoC defines the interconnection and behavioural semantics of models within the domain of the MoC [40]. In the previous subsection we came across a *continuous time dynamical systems* MoC and a *signal-flow* MoC. Typical MoCs used in engineering and research are *finite state machines*, *petri nets*, *communicating sequential processes*.

Models of Computation generally describe the semantics of interconnection and behaviour of computer processes. We suggest extending this to include models of physical processes, such as continuous time dynamical systems, as well. This gives us a unified framework for modelling both physical and cyber processes. Being able to model physical processes in the same framework as computational processes is a great benefit for CPSs as it allows for a natural way to combine the cyber world and the physical world.

We call models in a specific MoC *processes*. The MoC provides interconnecting and behaviour rules for processes in its own domain. The MoC does not, however, provide rules for combining models of different domains. This makes interconnecting heterogeneous models a non-trivial task (as we showed in section 2.8). Yet, heterogeneous interconnecting is a core characteristic of CPSs and is of essential importance to modelling, simulating, synthesizing and validating them. Different methods to solve this heterogeneous interconnection problem exist.

We list four types of methods:

- Frameworks that wrap a model with an adaption layer that unifies the exposed MoC such as ForSyDe [41].
- Modelling environments that interconnect heterogeneous models through domain-specific receivers and directors such as Kepler [42] and Ptolemy II [43].
- Models that expose a common interface (MoC) and that are responsible for translation between their native MoC and the commonly exposed MoC such as Functional Mock-up Interface (FMI) [44].
- Modelling languages that natively support different MoCs like SystemC [45, 46].

The right method for a given situation is application dependent.

We claim that MoCs and processes are central to the design of heterogeneous systems like CPSs and that they should take a central role in an architecture as a way to formalize models, behaviour and interaction.

2.10 Components of a CPS

We proposed the use of MoCs and processes as a formal way to model the behaviour of CPS. This behaviour is realized by the system's components. We categorize these components in two categories: *cyber components* and *physical components*. Cyber components are *physical* components that form the *cyber* world: cyber-nodes, sensors, actuators, communication media and external nodes:

Cyber-Nodes The general CPS that this paper concerns consists of a finite number of *cyber-nodes*. Cyber-nodes are devices that provide *computing* resources, can *communicate* through communication media and may have *storage* facilities.

Sensors and Actuators Cyber-nodes connect to *sensor* and *actuator* devices that respectively sense and affect properties of the physical world. These devices form the interface between the physical and the cyber world. The physical embodiment of the CPS is part of the physical world as well, so the system might sense and affect its own state as well. For example, the CPS may measure its own temperature and acceleration. Sensor and actuator devices model relevant relations between the CPS and the physical world. Peripheral devices (e.g. I/O modules, webcams and HVAC systems) that connect to cyber-nodes provide both sensors and actuators. A motor-driver, for example, might actuate field-effect transistors and sense currents, temperatures and logical system states. The architect is free to choose a suitable level of abstraction for the output of sensors and input of actuators. Sensors and actuators cannot communicate with other components but cyber-nodes by definition. Configuration and data exchange between sensors and actuators occur through cyber-nodes. A *smart* sensor that connects to a communication medium, like a Wi-Fi connected climate sensor, is a cyber-node with one our more sensors.

Communication Media Cyber-nodes can communicate through communication media. Examples of media are serial buses, the air, electrical connections, a protocol-stack. The level of abstraction depends on the requirements of the application. A higher level of abstraction means that the communication medium has greater responsibility of managing the communication. The characteristics of communication (determinacy, delay, bandwidth, . . .) follow from the chosen medium. The designer should choose a medium that supports the systems concerns and requirements. This is generally achieved in an iterative way: the designer either chooses an ideal (no delay, perfect transmissions, . . .) our realistic existing medium (EtherCAT, Wi-Fi, . . .) and uses this medium to check, through formal analysis, prototyping our simulation, whether the system achieves its requirements. Depending on the outcome of this check, the designer can adjust the medium (cheaper with less bandwidth, more realistic by introducing delay, . . .).

External Nodes External nodes can interact with cyber-nodes in the CPS by joining a communication medium. External nodes do not interact through sensors and actuators but directly via the communication medium.

Physical components are components that form the relevant aspects of the *physical world* and include components that either belong to the system our to the environment like motors, robot links, a car, electrical networks, pneumatic mechanisms, dynamic systems, objects, persons, the universe. The native MoC of physical components is *continuous time dynamical systems* although it might be useful to model these components by MoCs with a higher level of abstraction. Figure 2.6 shows how a CPS can be expressed as a graph of cyber-nodes connected through communication media, sensors, actuators and external communication. Physical components are connected to sensors and actuators and to each other. White shapes visualize the vertices that represent physical components and the overlap of two shapes visualizes the edge that represents the connection of the physical components.

Let us show how graphs can describe the physical model of a CPS. A graph is a pair

$$G = (V, E) \tag{2.5}$$

with V a set of vertices and $E = \{\{x, y\}|(x, y) \in V\}$ a set of unordered pairs of elements in V. A graph that describes a CPS is a tuple

$$G_{\text{CPS}} = (V_c, V_s, V_a, V_e, V_m, V_p, E_{c\ sa}, E_{sa\ p}, E_c, E_p) \tag{2.6}$$

such that V_c, V_s, V_a, V_e, V_m, V_p are vertices of respectively cyber-nodes, sensors, actuators, external nodes, communication media and physical elements.

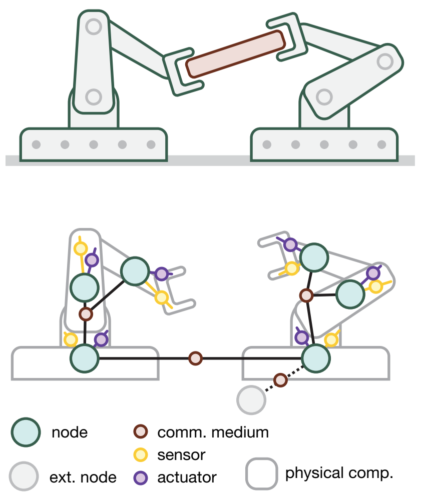


Figure 2.6 Cooperating robotic arms in a manufacturing environment. A graph overlays the schematic drawing to show how a graph of cyber-nodes, sensors and actuators can represent a CPS. Note that the white shapes of the robot arm represent vertices V_p whereas their overlap represents the edges E_p.

- $E_{c\ sa} = \{\{x, y\}|x \in V_c, y \in (V_s \cup V_a)\}$ are edges that connect cyber-nodes to sensors our actuators.
- $E_{sa\ p} = \{\{x, y\}|x \in (V_s \cup V_a), y \in\in V_p\}$ are edges that connect actuators our sensors to physical components.
- $E_{m\ ce} = \{\{x, y\}|x \in V_m, y \in (V_c \cup V_e)\}$ are edges that connect communication media with nodes.
- $E_p = \{x, y \in V_p\}$ are edges that connect physical components to each other.

This graph represents the components of a CPS. From the definition of edges, we can see that every path from a physical component to a cyber-node contains at least one sensor our actuator. This corresponds to Figures 2.2 and 2.3 in which sensors and actuators form the interface between the physical and the cyber world.

Strictly speaking, cyber components are part of the physical world as well: it is possible that the physical manifestation of a cyber component, like a sensor, influences the behaviour of the physical components. In that case, the designer should add a physical component that accounts for the influence of the component. For example, a voltage sensor in an electrical circuit does ideally not influence the behaviour of the system effectively. However, when it does, the designer can model this effect as an impedance parallel to the ideal sensor. This approach ensures that the cyber and physical worlds remain separated and Figure 2.7 visualizes this principle.

An alternative categorization of components was proposed by Bhave et al., consisting of: data stores, computation and I/O interfaces in the CPS domain; energy storage, sources, dissipative components and physical transducers in the physical domain; cyber-physical interface components and cyber-physical interface connectors [47].

This method combines the system interconnection structure with that of the interconnection of physical components. An advantage of this approach is that it provides overview in smaller systems. A downside

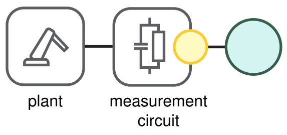


Figure 2.7 When a sensor our actuator has an effect on the physical components of the system it should be modelled as a physical component that represents this effect parallel to the CPS sensor our actuator component.

Table 2.1 Comparison of Bhave et al.'s components and the components presented in this section. Motivations of differences with respect to Bhave et al. are given.

Bhave	**This Work**	**Comments**
Cyber		
Data Store	Cyber-/external-node	Data storage, computation and computational transformations are part of the cyber domain and therefore belong to cyber-nodes and external nodes.
Computation	„ „	
I/O Interface	„ „	
Communication	Communication Media	The link between nodes is a concern of components, represented by communication media, the way of interacting is not a component concern.
Physical		
Energy Source	Physical Component	Physical components represent systems with corresponding behaviour. This behaviour can be specified in many ways of which the energy based approach is one.
Energy Storage	„ „	
Energy Dissipative	„ „	
Physical Transducer	„ „	
Common variable	Edges E_p	The connection between physical components is represented by edges between them. The behaviour of the connection is not a component concern.
Common effort	„ „	
Measurement	„ „	
Cyber-Physical		
CPI components	Sensors / Actuators	The interface between cyber and physical is represented by sensors and actuators which might be accompanied by physical components to account for their physical influence.
CPI connectors	„ „	

is that it enforces a specific set of physical components with corresponding behaviour: an energy-based modelling domain. Also, this approach dilutes the separation between structure and behaviour. The set suggested earlier in this section does not enforce specific physical components.

For comparison, elements of Bhave et al.'s approach are mapped to this work's components in Table 2.1.

Validating the usability of the proposed categorization and corresponding denotation of components is important. The use cases will demonstrate the proposed abstraction which is then evaluated in the discussion.

2.11 Summary

To be able to propose an Architecture Framework (AF), the characterizing properties and elements of a CPS need to be known. In this chapter, an analysis of general CPSs was conducted to distil these properties and elements.

Two worlds were discussed, the physical world and the cyber world.

The cyber world contains information about the physical world that the CPS uses to achieve its designed goals. A CPS connects the two worlds through sensors and actuators.

By definition, CPSs involve in interactions between the cyber world and the physical world. The CPS investigates and interacts with the complex physical world. Three modes of investigation were mentioned, all relying on models. Models are an essential component of CPSs. They are useful for testing, simulating, code generations, state estimation, prediction and more. The rules and semantics of a model are described by MoCs. As CPSs are heterogeneous, different MoCs need to interconnect and this is non-trivial. Solving this problem is a job that academics and industry now undertake and that results in possible methods that enable interconnecting heterogeneous models.

A simple and somewhat academic example that exposes most of the discussed properties is a CPS that optimally controls a physical subsystem. The CPS needs sensors and actuators to *investigate* the physical subsystem through *experiment*. For simplicity's sake, we assume that the system in this specific example is well-described by an Linear Time-Invariant (LTI) dynamical system and that the sensors and actuators are such that the system is observable and controllable. A linear-quadratic regulator is a controller that optimally controls the system in the sense of lowest cost. A weighted integral of quadratic error and effort determines the cost. The optimal controller requires full state feedback [48]. Although the full state is generally not available, a state observer approaches the actual state of the system as the system's model better represents the real system, which is the case for many practical systems looking at the omnipresence of state observers in real world systems. The estimated state with the optimal feedback gain, calculated from the system model with the algebraic Ricatti equation, yield the optimal controller: *theory* and *computation*. The controller states the actuator outputs which in turn actuate the system. This forms a closed loop. A realization of the controller typically (but not necessarily) runs on a digital system and requires discretised models and a Discrete Time (DT) MoC.

The following chapters construct the Architecture Elements (AEs) (section 1.6.1) of the framework based on the presented rationale.

3 Stakeholders and Concerns

CPS development is difficult because of the great number of dependencies that arise from the stakeholders and concerns through the system life cycle. The AF provides a structured way of working with stakeholders and concerns that helps to untangle and structure these dependencies. The goal of a stakeholder-concern analysis is determining why the system has the right to exist in its form in the first place. Its result is a set of stakeholders and per stakeholder a set of their concerns. Stakeholders have interest in a system because it addresses their concerns. Figure 3.1 visualizes this.

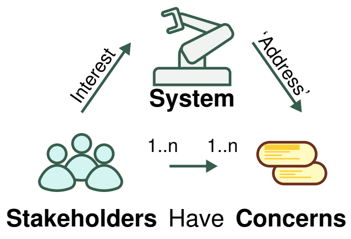


Figure 3.1 Interest in a system comes from stakeholders due to their concerns being addressed by that system.

This chapter provides a set of *domain* stakeholders and *domain* concerns, stakeholders and concerns that are common to the CPS domain. When creating an AD, the user of the framework extends these sets to incorporate application specific stakeholders and concerns. These extra concerns typically boil down to those that involve functionality and business as these are specific to the problem.

3.1 Domain Stakeholders

The AF provides a selection of stakeholders that involve in a typical CPS: the *domain* stakeholders.

This selection bases on the ISO/IEC/IEEE 42010:2011(E) suggested set of stakeholders for generic systems. The standard suggests users, operators, acquirers, owners, suppliers, developers, builders and maintainers. From this set, three core stakeholder groups are selected that are present in any CPS: those that **design**, those that **construct** and those that **use** the system.

Architects, Designers and Developers , constituting the *designing* group, determine requirements and translate them into actual software and hardware components or blueprints thereof. They might work in teams and be responsible for whole or part of the system. Their involvement concentrates around the design and reuse phase of the CPS life cycle.

Builders and Maintainers , constituting the *constructing* group, construct and maintain the system. They distinguish from *architects, designers and developers* in that they are not responsible for design decisions. They work with the system but distinguish from end-users by the fact that they are more experienced and have in-depth knowledge of relevant aspects of the system. They are mainly involved in the manufacturing, assembling and maintenance phase of the CPS life cycle.

End-users , constituting the *using* group, are the intended users of the system. Their involvement is in the operational phases of the life cycle. They care about whether the system is useful in completing its goals.

These three groups are *logical* stakeholder groups and a physical user might belong to more than one group. For example, if a person both *constructs* and *uses* the system, they are member of the *constructing* and the *using* group. The user of the AF remains free to add other, more specific, groups to their application specific AD.

The question remains whether this selection of application unspecific stakeholder groups is appropriate. The use cases will provide some insight in whether this is the case.

3.2 Domain Concerns

Concerns describe the interests of stakeholders in a system. The AF proposes concerns that represent fundamental interests of stakeholders in a CPSs, the *domain* concerns, like modularity and simplicity. To this set, the user of the AF adds application specific concerns that describe the specific interests in the system under consideration. Section 3.3 explains this extensibility of concerns in more detail.

The domain concerns that the AF proposes are:

Effectiveness The effectiveness of the system refers to the extend to which it achieves its goals. The architecture framework should support the design and management of effective systems. It should provide tools for the analysis, design and realization of corresponding functionality. Effectiveness is important to the architects, designers and developers during the analysis, design and realization phase and important to the end-user to allow its concerns to be addressed appropriately.

Modularity The key to complex systems design is modularity. Simple components can bring forth complex behaviour when interconnected [37]. A limitation of humans is that they cannot comprehend complex systems from all relevant viewpoints at once. Architects, designers and developers should be able to develop complex systems in a modular way such that they can focus on a specific aspect at a time. Builders and maintainers benefit from modularity as this can make repairing and constructing the system easier. In a software context, modularity can furthermore reduce code duplication and promote code-reuse. Separation of the different aspects of a system is known as Separation of Concerns [49]. Figure 3.2 visualizes modularity in a general way: by specifying clear interfaces, components can be optional and interchangeable.

Consistency A system and its architecture should be consistent such that the relationships between different views and models are valid [50]. Ensuring consistent relations promotes the early discovery of potential integration problems. This is in the interest of architects, designers and developers during development and validation of the system. Builders and maintainers rely on the consistency of views as well, when constructing a part of the system.

Reusability The typical software engineering principle that promotes reusability is *Don't repeat yourself (DRY)*. Reusability applies not only to the life cycle of a system but also to other related systems. Reusing components in different applications is beneficial for the stakeholders concerned with development and construction: it reduces development time and it promotes understandability and reliability by throughout testing and documentation of high-quality modules.

Extensibility The requirements of a CPS can adapt to changing needs during its lifetime. To support the addition of functionality, a system should be extensible. This is especially important when end-users' needs change. The implications of extensibility differ per context. Figure 3.2 shows extensibility in an abstract context of *components*. In a stakeholders-concerns context, extensibility means the ability to freely add stakeholders and concerns. Section 3.3 explains this kind of stakeholder-concern extensibility. In a software context, extensibility means the use of buses and clearly defined interfaces [51]. Extensibility in the context of specific viewpoints is discussed in chapter 4.

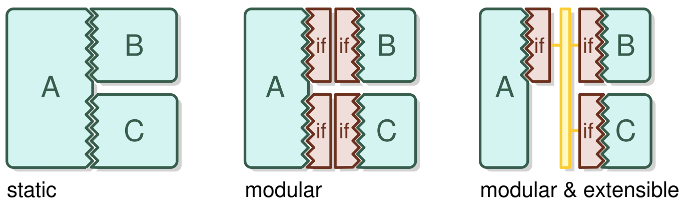


Figure 3.2 Abstract view of modularity and extensibility of three components. Interfaces ensure compatibility between components such that they can be modularly used and exchanged. A *bus* is a means to connect multiple components through a standardized interface. These principles hold for hardware, software and architecture.

Testability Concerns the validation of the operation of the system with respect to the expected behaviour and its intended goals. Testing can reduce the development risk and efficiency by early detection of problems with the system. Tracing the source of an error is easier when tests are automated and targeted towards specific functionality, (automated) unit tests serve this purpose. Validating the system and its features as a whole is complementary to these unit tests and is best handled by feature or acceptance tests. Testability of a system is of interest to all that design, develop and manage one and is therefore an important domain concern.

Table 3.1 Stakeholder-Concern traceability table

Concerns	**Effectiveness**	**Modularity**	**Consistency**	**Reusability**	**Extensibility**	**Testability**	**Understand-ability**	**Simplicity**
Archi., Design.and Devel.	✓	✓	✓	✓	✓	✓	✓	✓
Builders and Maintainers		✓	✓	✓	✓	✓		✓
End-users	✓							

Understandability Understandability refers to the degree to which a system is understood by people with relevant background knowledge of the systems fundamental principles. By integrating the architecture description in a system, it contributes to its own documentation and explanation. The ultimate form of understandability is a system that explains itself to anyone with appropriate background knowledge. A nice example of how a system's description follows its function is the bond-graph [52], assuming fundamental background in bond-graphs. The visual structure of the bond-graph model represents the physical form and function of the system it describes and it integrates this into the actual system model without introducing additional documents or models. Understandability is of great importance to architects, designers and developers and to a lesser extend to builders and maintainers.

Simplicity Keep it short and simple (KISS), is a well-known principle in software engineering introduced by Edger Dijkstra. The designer should try to keep an already complex system as simple as possible to avoid introducing unnecessary dependencies that limit the design process.

Table 3.1 is the *stakeholder-concern traceability table* that shows which stakeholders have interest in which concerns.

This selection of general application-unspecific concerns should frame most of the CPS development issues. In chapter 6, use cases are elaborated to demonstrate whether the AF indeed helps in addressing these issues and correspondingly helps in managing the CPS design and management complexity.

3.3 Using Stakeholders and Concerns

This architecture framework specifies a starting point for stakeholders and concerns. This starting point targets interests in the general lifecycle of a CPS but not the application-specific interests. Both are of importance but as the application is not specified in advance this framework considers application unspecific concerns. The designer should extend the starting point to suit the actual application stakeholders and concerns. This means that the designer may regroup, change, add or remove stakeholders as well as concerns and the relations between stakeholders and concerns. Figure 3.3 visualizes this.

A formal way of describing the stakeholders and relations is a bipartite graph. Such a graph has a mathematical representation:

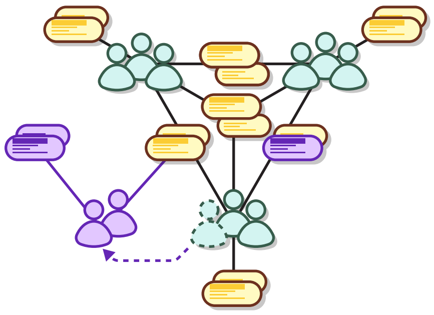


Figure 3.3 The framework stakeholders and concerns (green and yellow) are the starting point for an application. The designer should add application-specific stakeholders and concerns to this starting point. This figure shows how the designer can add extra stakeholders and concerns and change an existing concern.

$$G_{\mathrm{S\&C}} = (V_S, V_C, E_{SV}) \tag{3.1}$$

with V_S and V_C sets of vertices that respectively represent stakeholders (listed in section 3.1) and concerns (listed in section 3.2) and $E_{SV} = \{(x, y)|x \in S, y \in C\}$ a set of edges that are pairs that relate stakeholders to concerns (these edges correspond to the checks of Table 3.1).

The user of the framework can use this graph to determine how concerns and stakeholders relate to each other. Automated procedures can read this graph to analyze relations between stakeholders, concerns and other AEs which can provide the user of the framework with more insight.

3.4 Summary

This section introduced stakeholders and concerns that are generally involved in the design and management of CPSs. Stakeholders are reduced to three groups: those that design, those that construct and those that use the system. Eight stakeholders were described. Together, these serve as the starting point and can be extended with stakeholders and concerns that are appropriate to a specific system.

4 Viewpoints

In his 1982 note *on the role of scientific thought*, Dijkstra described what he then called the *separation of concerns*. This idea started from the notion that a *good* system must conform to a wide range of requirements that yields a correspondingly complex design problem. Dijkstra noted that, though the system must conform to all of them simultaneously, studying all aspects and requirements simultaneously is not an effective approach. Instead, he argued, studying a system from a single specialized perspective at a time allows better understanding and more effective analysis [53].

Viewpoints provide these perspectives that allow the designer to focus on a part of the problem without entangling themselves in the full problem. The downside of this approach is that viewpoints could diverge and conflict. The AF provides architectural tools (correspondence rules) and methodological tools (an iterative process and testing) to help the framework's user to avoid and resolve inconsistencies.

Furthermore, viewpoints provide documentation structure that guides users in a focussed manner through an architecture. So, an AD decomposes the architecture of a system in views using viewpoints that each explain a subset of the aspects of the system of interest.

The AF - described in this document - helps users in decomposing CPSs from 5 fundamental viewpoints. Figure 4.1 shows the structure of the proposed framework. The figure introduces the base 5 viewpoints and viewpoint support.

The viewpoints are as follows:

Scenario Viewpoint concerns the expected interaction of the system with its environment, based on the stakeholders' concerns that are involved in the design. The scenarios should capture what stakeholders want of a system and relevant subsystems but not how these goals are achieved. Scenarios are discussed in section 4.1.

Logical Viewpoint decomposes the concerns and scenarios into a hierarchy of objectives accompanied by qualities and metrics of these objectives: goals and requirements. The hierarchy covers how the system's goals are achieved through objectives and requirements, and why, through corresponding decisions and rationale. The logical viewpoint is subject of section 4.2.

Process Viewpoint consists of models that describe the behaviour of

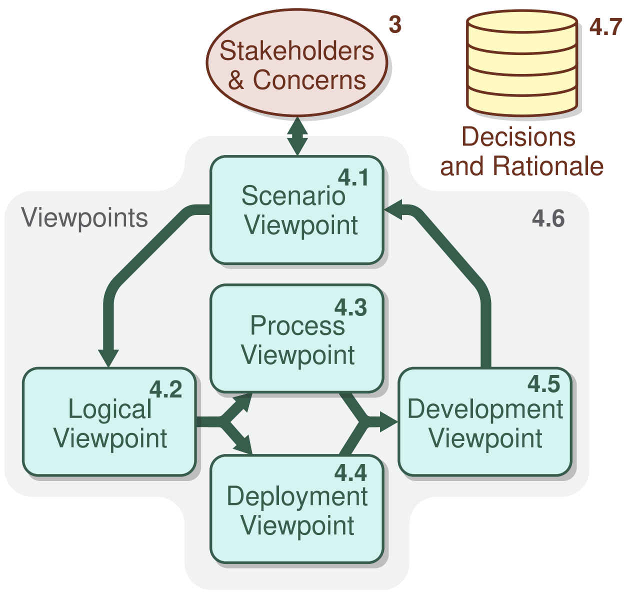


Figure 4.1 Structure of the viewpoints of the architecture framework. The arrows describe the typical order of elaboration of viewpoints, starting with the set (oval icon) of stakeholders and concerns. The order forms a cycle: the development viewpoint closes a single iteration and provides input to the next. Scenarios, simulations and tests are evaluated and compared to the system's concerns. This provides input to a new cycle to improve the architecture of the system. A database (stack icon) of decisions and rationale contains the fundamental reasoning behind main choices of the system.

the system - both cyber and physical - to realize the objectives and requirements defined in the logical viewpoint. The process viewpoint is discussed in section 4.3.

Deployment Viewpoint concerns the physical connections, or structure, of the system - both cyber and physical - to realize the objectives and requirements of the logical viewpoint. Section 4.4 discusses the deployment viewpoint.

Development Viewpoint concerns the implementation of the cyber-nodes of the system. This includes determining what the cyber-nodes execute and what components they require, which sensors and actuators need to be connected and what communication links supported. The result consists of one or more models that describe the blueprint of the different types of cyber-nodes. The development viewpoint is discussed in section 4.5.

Figure 4.1 also shows AEs of the framework that support the previously listed viewpoints. These elements are:

Stakeholders and Concerns As introduced in the previous chapter, chapter 3, stakeholders and concerns describe who is interested in a system and what their interests encompass.

Relations Correspondence and consistencies model the relations between the elements of the architecture. They describe how views should relate to each other. Correspondences and consistencies are introduced with every viewpoint and an overview is presented in section 4.6.

Architecture Decisions and Rationale Architecture Decisions encompass choices that affect elements of the architecture, justified by rationale. The relevance and importance of recording and organizing rationale and decisions has been stressed by architects that are heavily involved in the evolution of architectures in engineering [54, 55, 56]. These decisions might raise new architecture concerns or dependent sub-decisions. Architecture decisions are introduced in section 4.7.

Though the set of viewpoints bears naming resemblance to Kruchten's 4+1 view model, the proposed framework is not a mere translation of Kruchten's. Rather, the high-level separation of concerns of the 4+1 framework was found applicable to CPSs too. The proposed framework is designed from the ground up for CPSs and the viewpoints themselves and their order vastly differ from the 4+1 view model. Section 7.4.2 expands on the differences from other frameworks.

In the Use Cases section, we apply the viewpoints to three cases and analyse whether the chosen set of viewpoints indeed helps in managing CPSs throughout the life cycle.

We will now discuss the viewpoints in more detail. Each viewpoint includes:

- Stakeholders and Concerns
- Model Kinds
  - Language/Notation
  - Metamodel
  - Operations on Model Kind
  - Correspondence Rules
- Operations on Views

Viewpoint correspondence rules are provided in section 4.6, after the introduction of the viewpoints. Examples and sources are provided throughout the text.

4.1 Scenario Viewpoint

It all starts with knowing what you actually want and the scenarios viewpoint is the appropriate way to find out. The scenarios view plays in important role in the iterative development of a system. It provides use cases and scenarios that describe and help discover the goals of the system at the beginning of an iteration while serving as a means of integration and testing at the end of an iteration. The usefulness of scenarios is two-fold:

- Scenarios drive the discovery of architectural elements of the AD of the CPS during the design phase.
- Scenarios provide documentation, example, illustration and tests of the system under consideration. They provide a means of communication and help the stakeholders to understand the system in its environment.

The scenario viewpoint concerns the purpose and the intended operation of the system from the stakeholders' perspective: the 'visible' interface between the system and its environment. Describing in what way this operation is achieved is not a primary concern of this viewpoint but that of the other viewpoints. At the end of a development iteration, however, the outcome of the other viewpoints is integrated in the scenario models to assess and validate the system.

The designer can create the scenarios by careful evaluation of the application-specific concerns. The architect can use the feedback from the validation of the architecture using scenarios to revise and improve the requirements. This is the basis for an iterative development process.

Scenarios can be designed using different kinds of models, depending oon n the context of the system. A suitable model is able to express the intended and unintended interaction of the system of interest with its environment as to operationalize the stakeholders' concerns. Subsection 4.1.2 elaborates on what makes a model kind suitable for scenario specification.

This section introduces two examples of suitable models: *SysML/UML-2* use case and activity model kinds; *multi-body dynamical systems* model kinds. But first, it explains the relevance of this viewpoint to a CPS' stakeholders in the context of their concerns.

4.1.1 Stakeholders and Concerns

The scenarios viewpoint is by design of interest to all stakeholders. It helps in defining system requirements, documenting and testing. The scenarios help validating the implementation of the architecture using simulations or co-simulations with hardware in the loop.

Effectiveness The scenarios view frames the effectiveness concern by connecting the concerns of the system to typical usages. These scenarios help in analysing how the system can effectively fullfill its purpose. The iterative development method that the scenarios view supports helps the designer in focussing on the effectiveness of the architecture without getting lost in the details of the other views.

Consistency The designer can derive integration-tests from the scenarios that help in detecting inconsistencies in the design.

Extensibility Addition or alteration of system requirements directly relates to the corresponding scenarios. The designer can see the impact of changing or adding requirements by checking how the relevant scenarios and use cases relate to other views. The scenarios form a clear starting point for extending the system architecture.

Testability The designer can use the scenarios for integration- and acceptance-tests. These tests are also useable as regression-tests to ensure the correct functioning of the whole system during development.

4.1.2 Model Kinds

Different kinds of models are suitable to specify the scenarios of a system. The suitability of model kinds for scenarios depends on the system of interest. In Software Intensive systems, for example, sequence diagrams model expected scenarios. For CPSs, scenarios need to take into account the interactions between the physical environment and the system too.

User stories and *multi-body dynamical systems* are typical examples of suitable types of models. The following text introduces two groups of model kinds to model scenarios: SysML/UML-2 based activity and use case models and multi-body dynamical system models.

Selecting the right model(s) for the job is a task for the user of the framework, guided by their experience and insight. Often, this is a matter of what models and tools the users of the framework feel comfortable with; the availability of knowledge and tools; successful use of models in other projects. The selection of the right model remains a, possibly very difficult, task for the user of the framework.

The set of model kinds that can be used in virtually infinite, making it impossible to discuss each and every one. Instead, the framework allows the use of additional model kinds under the condition that their description is included or referred to in the AD in accordance with IEEE 42010:2011 B.2.6. This allows users of the AD to use, analyse and interpret these additional model kinds like those mentioned in this AF. In general, the model must be able to express the intended interaction between system and environment with respect to the stakeholder's concerns. For example, if a concern is that the systems succeeds in a certain sequential interaction between the system and an end-user, a suitable model kind might be a SysML/UML2 activity diagram. If the concern is that the system must undergo a dynamical physical interaction with itself and its environment, a multi-body dynamical system might describe the scenario well.

4.1.3 Model Kind: SysML/UML2 use case and activity

The UML2 specification [57] defines a number of behaviour constructs that can be used to express the behaviour of systems. These constructs are activities and use cases and they are designed to model a diverse and wide range of systems. State machines and interactions, other UML2 behaviour classes, are less suitable for scenario description and more for detailed description of behaviour in the process view. The UML2 standard describes the meta-model of these models, the templates and languages for these models and the operations on them.

Note that SysML, a standard for systems modelling, reuses these UML2 meta-models so they correspond to SysML too [58]. These SysML/UML2 constructs are briefly introduced below:

Activity Models

Also called *control flow* or *object flow* is a sequence of actions that aims at modelling the sequence and conditions of behaviour. Figure 4.2 shows an example of an activity diagram that models a scenario of a set of coordinating robots. This diagram is based on a set of concerns: the

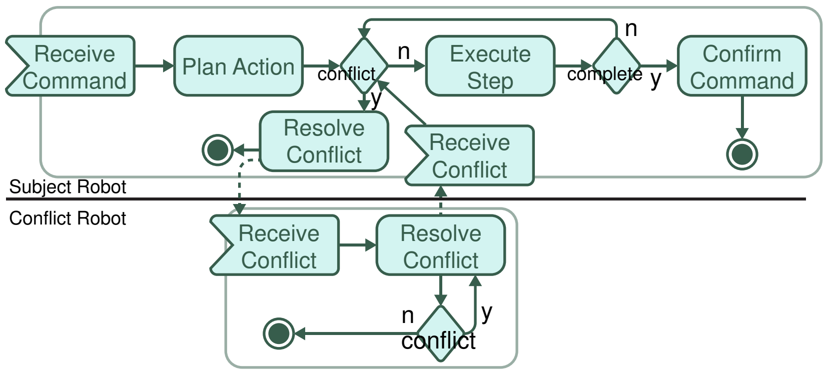


Figure 4.2 Activity Diagram of Coordinating Robots

robots must accept commands; the robots must autonomously avoid conflicts. It describes how the system should interact without specifying the implementation.

When a prototype, simulation or logical model of the system is available, the model can be used to assess the system by executing the described sequence of actions and check whether the system adheres to the scenario. In this case, the scenario can be tested by sending a command to the system and either inflict a conflict or not, to check if the system acts as intended. The result of this process should be discussed with the relevant stakeholders.

Use Case Models

UML2 introduces a *use cases* that consists of meta-models for specifying the required usage of the system. As such, the *use cases* package is an ideal candidate for supplying modelling semantics in the scenarios viewpoint of the AF.

Central to this package of meta-models are actors and use cases next to a number of supporting meta-models (classifier, extend, etc.).

Actor An entity that interacts with the subject and is external to it.

Use Case An entity that models interactions within a system with observable results.

A specific use case may *own* other behaviour constructs - such as activities, interactions and state machines - that define how the subject interacts with the actor.

Actors in these diagrams may directly point to stakeholders in the AF. A use case diagram clarifies who has interest in specific behaviour and motivates them to exactly specify their requirements. Use case diagrams describe the use cases only briefly so they need to be accompanied by other diagrams. An example use case diagram for top-level use cases of a Unmanned Aerial Vehiicle (UAV) is shown in Figure 4.3.

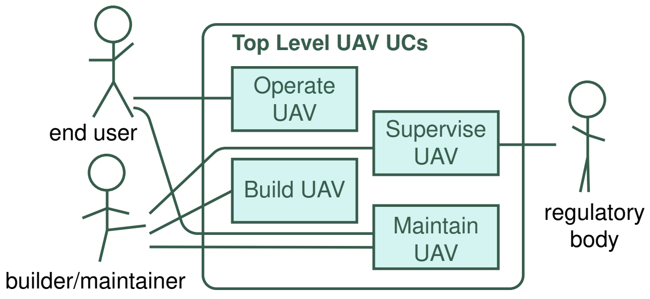


Figure 4.3 Use Cases diagram of a UAV

4.1.4 Model Kind: Multi-body Dynamical Systems

Multi-body dynamics models can typically mimic the operation of a real-life system or subsystem and its environment through computer-simulation.

They can provide insight in the (intended) behavior of such a system and environment. Constraints on the operation of the simulation may help to estimate performance, indicate issues and test the system.

Modern simulation tools and model repositories allow the fast and effective prototyping of systems and their environment for the purpose of providing scenarios. Examples of scenario models that are suitable for CPSs' design are MATLAB, Gazebo and 20-Sim models.

4.1.5 Operations on Views

Operations describe how the product of this viewpoint, the scenarios view, can be used.

Creation

Scenarios are created by evaluating the stakeholders' concerns and motivation. Typically, the user of the framework engages in a discussion with other stakeholders, sketches a scenario and has the stakeholder review it. At least some of the other stakeholders should be able to understand the scenario. All stakeholders should review other stakeholder's scenarios too and any inconsistencies in expectation that are clear before hand should be resolved.

Evaluation

The logical, process, deployment and development view will lead to an architecture description that can be integrated in the initial scenario to evaluate, assess and validate the result. For example, if a scenario describes an intended sequence of interactions between the system and a user, the resulting system architecture can now be used in this scenario to test whether the system passes or to assess the performance of the system.

When a situation matches a scenario but the realized (physical, simulated) system deviates from expectations, two causes are possible:

- The system is incorrect
- The scenarios are incorrect

The actual cause could be a combination of these too. Either way, something went wrong and a resolution should be sought. Both causes will occur during the life cycle of a system as specifying scenarios can be as difficult as designing a system and both may supply useful insights in the systems of interest.

Typical resolutions are:

Modifying the system This will lead to a chain of consequences involving the system's viewpoints.

Modifying the scenario Which might lead to a revision or reinterpretation of concerns and requirements.

Accepting the inconsistency When the inconsistency is of minor importance. The inconsistency should be tracked down, recorded in the AD and motivated.

4.2 Logical Viewpoint

The concerns and scenarios are important in translating the stakeholder's wishes to achievable requirements. The logical viewpoint focusses on the decomposition and grouping of functionality in the form of a hierarchy of building blocks. Its goal is to ensure the effectiveness of the system of interest with respect to its concerns. The logical viewpoint helps in defining high-level systems goals and creating a hierarchy of functionality from these goals.

The uses of this viewpoint are:

- Serve as a framework for reasoning about the functionality that the system should provide to be effective.
- Promote separation of application dependent functionality from more general functional building blocks.
- Discover the functionality and components that are needed to achieve the intended system goals.

The logical architecture view describes the core functional elements of a system and the dependencies of these functional elements: the building blocks of the system. Accompanying these blocks are relevant requirements on their operation such as minimum bandwidth, cost, technical implementation and development effort.

First, the relevance of this viewpoint to a CPS' stakeholders and their respective concerns is discussed. Then, the *functionality diagram* is introduced.

4.2.1 Stakeholders and Concerns

The insight in functionality and decomposition from the viewpoint is important to all stakeholders. It bridges the stakeholders' functional concerns with elements of the system.

Effectiveness The logical viewpoint helps in defining the functional components that support the application goal. The logical view relates requirements and concerns to components. When requirements or concerns change, the logical view shows which components change accordingly.

Modularity The logical viewpoint promotes explicit functional modularity. The modularity of functional components allows developers of different specializations to work on the system concurrently.

Reusability The separation of concerns that also benefits the modularity promotes the independent reusability of functional components.

Understandability The logical viewpoint explains how functional components relate to each other.

4.2.2 Model Kind: Functionality Diagram

The central model kind of this viewpoint is a hierarchical diagram of functional elements.

Metamodel

The classification of these elements is based on the *Separation of Levels* as used in the EU RobMoSys project [59]. The classification consists of the following classes, the totally ordered set S enumerated by ordinal numbers.

S_1 **Goal** The high-level goal of the system, the functional stakeholder concerns, regardless of the implementation.

S_2 **Task** Decomposition of a goal into tasks that the system must achieve. Describes a composition of skills.

S_3 **Skill** Basic functionality of the system, the building block of tasks. Skills rely on a set of services.

S_4 **Service** The access-point of system functionality. Services should hide their implementation from other parts of the system and provide an interface instead. This makes them the preferred separation between systems and sub-systems.

S_5 **Function** Functionality that the system provides to services. Implemented by code, a model of computation, a library. The lowest functional level.

Each functional element belongs to one of the classes and relates typically to elements in a higher class (supports) and elements in a lower class (depends). The elements may have one or more requirements that define metrics and characteristics that the functional element should achieve to be *successful*. Requirements are metrics that indicate the qualitative and quantitative properties that a logical element should adhere to.

Each element of the hierarchy configures and coordinates the lower level elements such that it can achieve its goals. That means that the elements of the hierarchy include some logic to orchestrate its supporting elements. In this way, the supporting elements do not have to know about their exact use and configuration in the hierarchy which reduces the coupling between components and improves reusability and modularity.

Language

The hierarchical diagram is an undirected graph $G_{\text{logical}} = (V, E)$ with V the set of vertices that has a type through the mapping $\text{type} : V \to S$, a name through $\text{name} : V \to N$ where N is the set of names and an annotation through $\text{annotation} : V \to A$ where A is the set of annotations including the empty annotation; E the set of edges $E = \{\{x, y\} | (x, y) \in V \wedge x \neq y\}$: unordered pairs that connect different vertices. The graph is undirected because the ordering of vertices - and consequently the direction of the edges - follows from their level $(S_N | N \in \{1..5\})$. In most cases, an edge connects adjacent classes: the difference between the two index numbers of the types of the vertices $\text{type}(v) \in S$ is one or occasionally zero (normally elements on the same level connect through a coodinating element at the higher level). This allows a straightforward partition of the graph into regions in which all vertices are of a single type. If this is not the case, an annotation should include motivation and the vertices should have individual graphical labels that denote their type.

Figure 4.4 shows a graphical denotation of the information contained in G and the mappings type, name and annotation. These diagrams provide a graphical language that framework users can use to document the logical view.

4.2.3 Operations on Views

Creation

Creating the view requires a list of concerns and a set of scenarios. The designer constructs the model by distilling goals from the concerns and the scenarios, tasks from the goals, and so forth. If specific requirements need to be fulfilled, these can be added to the elements. The requirement might be associated with an architecture decision with corresponding rationale. Architecture decisions are discussed in section 4.7.

Figure 4.4 shows an example instantiation of this model.

Grouping the elements of the view allows for the creation and composition of complex logical views.

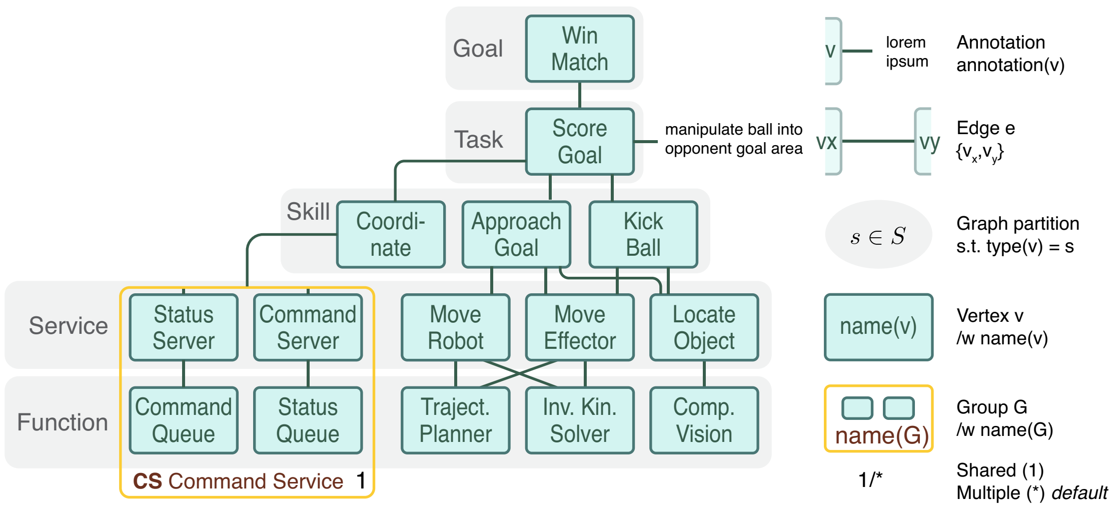


Figure 4.4 Example instantiation of the diagram for a robotic soccer player. Note how the elements are partitioned in goals, tasks, skills, services and functions. A feature for the sake of overview is *grouping* that allows parts of the diagram to be repeated or documented elsewhere.

Updating

During the development of the system the logical view will need updating to refine the system, improve its performance or resolve problems. Adding or refining functionality is a matter of adding or replacing corresponding logical elements and requirements. The relation between the the logical view and the other views helps in determining the impact of such changes. Situation may arise in which another view indicates issues which cannot be resolved in an acceptable way. In these situation, the corresponding logical elements and requirements might need revision of the logical elements or adjustment of requirements to allow a different solution. Correspondences between the logical view and other views help in tracing this relation.

Testing

The logical view elements can be accompanied by requirements that should be assessed at the end of a development iteration. Some requirements can be assessed using models in a single view (like: communication must be wireless), others require the integration of the views at the end of the iteration (like: the system should process at least 100 items per hour; be available 99% of time). When a model or system fails to satisfy a requirement, the requirements or the system has to be adjusted, or both.

4.3 Process Viewpoint

The process viewpoint describes the behaviour, flow of information, concurrency and distribution of the system including behaviour of relevant components of the physical environment. Model of Computation (MoC) is an important ingredient of the process view that provide a means to account for CPSs' characterizing heterogeneity. Choosing and composing MoCs allows the designer to select the appropriate abstraction level for the job. The uses of this viewpoint are:

- Serve as a framework for reasoning about the behaviour that realizes the functionality of the logical view.
- Describe the structure of different behavioural modalities by composing MoCs.

MoCs traditionally define the interaction and behaviour semantics of computer objects called actors [42, 16]. Most of the existing MoCs play a role in computer system analysis, synthesis and verification. Finite State Machine (FSM) and Kahn Process Network (KPN) are examples of such MoCs that are synthesizable to code or hardware. The designer should prefer reusing existing MoCs as a broad range of models is available in research, simulation tools and synthesis tools.

Though these models can play an important role in the design of cyber-nodes, a CPS design also needs to account for its (non-computer) physical elements. Lee and Sangiovanni-Vincentelli [40] demonstrated a framework for comparing MoCs, in which processes play an essential role: they specify the behaviour of a model of computation in the presence constraining inputs. Willems [39] described the *behavioural approach to open and interconnected systems* which describes physical dynamical systems from a similar behavioural point of view. The suggested solution here is that interpreting dynamical system models as MoCs unifies the cyber and physical parts of a CPS. The MoC concept can bridge the gap between physical and cyber processes which is useful for modelling and analysis.

4.3.1 Stakeholders and Concerns

The process view is relevant for the *architects, designers and developers* of the system. It helps in reasoning about the behaviour of the system and its environment.

Usefulness Models, in particular as processes in a MoC, are general concepts that provide a sound foundation to express a large range of behavioural modalities. Their composition is essential to any system. The trivial case, no composition, is a single MoC with a process.

Modularity The process view promotes splitting the behaviour of the system and relevant physical components in modules. Using a limited set of core processes to describe a more complex system is powerful and helps in keeping the system understandable.

Reusability Combining heterogeneous processes is possible through composition of MoCs. Thus, reusing processes in different contexts and systems belongs to the possibilities. This feature addresses the reusability concern.

Extensibility Composing MoC domains and processes allows natural extension of the system's behaviour.

Understandability The process view provides a global overview of the, potentially distributed, behaviour of the system. This is a property that benefits the understandability of the system. It

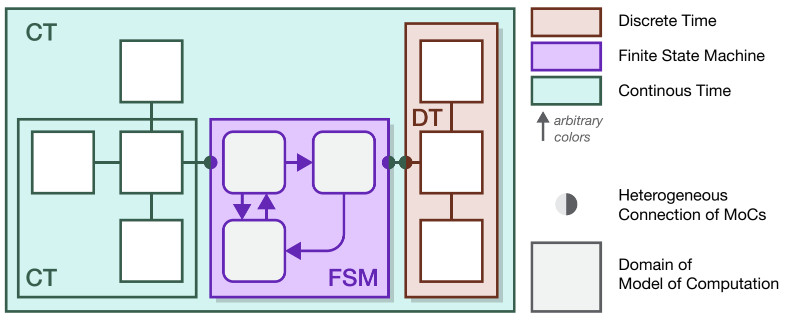


Figure 4.5 Example of model used in process view. The Continuous Time (CT) domain contains CT actors, a Finite State Machine (FSM) actor and a DT actor. The connections between actors are ports. The MoC or combination of MoCs determines the interaction semantics of the actors.

exposes relations between behaviours such that tracing how different behaviours relate is possible.

4.3.2 Model Kinds

The process viewpoint uses two architectural model-kinds and in a typical process view, both models will be used:

- Model of Computation, specifying operating and interconnection semantics of processes in their corresponding domain.
- Hierarchical composition of MoCs and processes.

The first model kind describes the model in its respective MoC domain according to the semantics defined by this model. We cannot provide an exhaustive list of possible models of computation as the framework does not specifically limit this list.

The second model kind used in the process view is a diagram of connected actors in hierarchically composed MoCs. This diagram shows how different actors connect in a specific MoC domain. Ptolemy II's notation of composed domains influenced this model kind [43]. Figure 4.5 shows a diagram that visualizes an example composition. The visual style of the figure is in no way fixed, so the user remains free to manage the model in an appropriate way. When designing a system, these models are typically created in a computer modelling environment that also visualizes the models.

MoCs are also useful to model the physical part of a system. Energy and power-based models such as bond graphs and Lagrangian models [60] can be used for this purpose. These models then represent the behaviour of physical components in the *deployment view*.

4.3.3 Model Kind: Models of Computation

Typical model kinds that can describe the flow of information through the system are MoCs such as Synchronous Data-Flow (SDF), Communicating Sequential Processes (CSP) and Kahn Process Network (KPN). CT and DT MoCs are often used in control systems.

Researchers and industry developed a broad range of different MoCs. A system designer can reuses an existing MoC if it suits his needs. Developing custom MoCs is another possibility but that topic is beyond the scope of this paper.

4.3.4 Model Kind: Composition of MoCs

Creating the model of composition of MoCs and processes starts with evaluating the logical view. The system designer must reason what behaviour the system must expose to support the functionality in the logical view. This process is highly application dependent. For inspiring examples, the reader is directed to section 6 that discusses the use cases.

After identification of the required processes, the designer can model relevant physical components of the environment in the process view as well. The natural MoC for these processes is Continuous Time (CT) as processes are dynamical models. Section 2.5 described the modelling and interconnection of these models and thus their MoC.

4.3.5 Operations on Views

Working with process views involves some operations: creating and updating, testing and realization.

Creation

The process view involves the behaviour and processes that enable the functionality of the logical view. A typical creation process starts by evaluating the top-most goals and how it needs to coordinate the corresponding tasks to achieve the behaviour that enables the goal. This is repeated for the elements of the task level in terms of skills and for the lower levels correspondingly.

This operation makes the problem of process design more comprehensible and understandable than trying to design all system processes simultaneously.

Updating

The view will need updating when the system's functionality changes or the processes do not achieve their goal. In the first cast, changes in the system's functionality are administered in the logical view. Through the relation between elements of the logical view and models in the process view, the user of the framework can determine which models need to be revised. In the second case, when processes do not achieve their goal, the corresponding logical elements can be traced back. Then, the user must decide upon the appropriate next step: changing the corresponding process, changing the corresponding logical element or changing both. The user can use the rationale and architecture decisions to help in choosing the most appropriate action.

Testing

The process view, or parts thereof, can be tested throughout the CPS life cycle. The appropriate tools and methods to do this depend on the used MoCs and their composition.

Realization

Some models will allow automatic realization in terms of executable programs through code generation and synthesis. This is generally the case for cyber MoCs and sporanically for physical MoCs. The realization of physical subsystems typically involves conventional engineering methods in the domains of mechanical, chemical and electrical engineering.

4.4 Deployment Viewpoint

The deployment viewpoint describes the embodiment of the CPS in its environment in terms of components and their connections. This viewpoint structures reasoning how the cyber world relates to the physical world in a CPS. The viewpoints provides help in determining the sensors and actuators that the CPS requires and what physical components and cyber-nodes they connect.

The scenario view, logical view and process view specified parts of the system that involve connections between the physical and cyber world of the system. These views, however, did not specify how the system senses from and acts upon the cyber and physical worlds. This is the main concern of the deployment view: design the connection of physical components, cyber-nodes, sensors, actuators and media to link them together and allow flows of information through the system to enable it to achieve its goal.

The idea is that the configuration of sensors, actuators and communication media should follow from the purpose of the system and corresponding processes, not the other way around. As sensors, actuators and communication are getting cheaper, smaller and omnipresent, it is more feasible to design the scenarios and processes without the constraints of sensor, actuator and communication availability. The configuration of sensor, actuator and communication is then designed afterwards. Still, limitations on the number, placement or performance of sensors, actuators and media might arise. These limits might prohibit an earlier defined certain process or scenario. Some of these issues can be resolved in the deployment view by changing the placement and configuration of sensors, actuators and communication. If the issue cannot be resolved in this view, the corresponding elements of the architecture must be modified in the current or a following iteration.

4.4.1 Stakeholders and Concerns

The deployment viewpoint is of interest to *Architects, Designers and Developers* and *Builders and Maintainers* through the following concerns.

Usefulness : The deployment view presents a general way to express the embodiment of a CPS in its environment. The models are general enough to apply to a broad range of systems. Yet, they provide the designer with insight in how the components of a CPS relate to the functionality and processes described in the logical and process view.

Consistency The deployment viewpoint is a hub that provides the designer with insight in how functionality, processes and cyber-node type models (discussed next) relate. This overview helps in keeping the system consistent. The correspondence rules discussed in section 4.6 further help in keeping the system consistent.

Reusability The deployment view promotes modularity of the system by identifying similarities between cyber-nodes. Cyber-nodes with many similarities can share the same development structure as will become clear in the next section.

Extensibility The deployment view improves the extensibility by providing a model that allows the addition of extra components and relations without fully redesigning the system. The designer can determine how a change in the embodiment of a CPS influences processes, functionality and implementation by evaluating the relations between these views.

Testability The deployment view enables trivial hardware-in-the-loop and software-in-the-loop testing of CPSs by either simulating or realizing a set of components in the model. For example, one can choose to simulate all components of an autonomous car but realize the cyber-nodes responsible for cruise control in hardware.

Understandability The deployment view is a natural way of describing the embodiment of a CPS because it can be modelled such that its components correspond to their actual location in the intended system.

4.4.2 Model Kinds

In section 2.10 we described a model of CPS components. We will adopt this model in the deployment view. This means that we represent the embodiment of the CPS and the relation to its environment by a graph of cyber-nodes, sensors, actuators, external nodes, media and physical components:

$$G_{\text{deployment}} = (V_c, V_s, V_a, V_e, V_m, V_p, E_{c\ sa}, E_{sa\ p}, E_c, E_p) \tag{4.1}$$

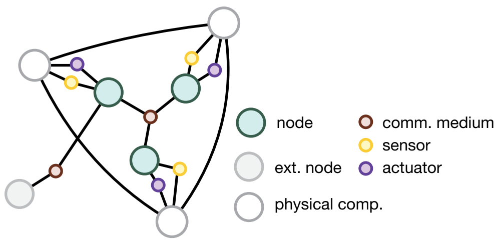


Figure 4.6 Visualisation of the graph model that represents the embodiment of a CPS in its environment. In this example, three cyber-nodes share communication medium and three sensors and three actuators interact with the environment which consists of three interconnected physical components.

of which section 2.10 discussed the meaning and interpretation. Figure 4.6 shows a visualization of an example graph.

This model is created by determining what the relevant physical components are that the CPS interacts with, what properties of these physical components need to be measured or influenced, what sensors and actuators are needed to achieve this and how these connect physical components with cyber-nodes. Secondly, the designer should determine which cyber-nodes need to exchange information and add communication media correspondingly.

The deployment diagram of a large system comprises many components in repeating patterns. The deployment diagram includes two features to help the user organize it. First, the user can place a (schematic) background image in the diagram. Second, the user can use named modules that unravel the diagram and allow for reuse of arrangements of components. Figure 4.7 shows a diagram that includes both features.

A module is a pair consisting of a deployment graph with an ordered list of externally accessible (public) vertices P. In case of the gripper:

$$M_{\text{gripper}} = (G_{\text{gripper}}, P_{\text{gripper}}) \tag{4.2}$$

Embedding a module in a deployment diagram is a matter of placing a reference to the module in the graph and connecting vertices to the publicly accessible vertices, indicated by ordinal number enumerated connectors. Vertices connect to these connectors like they do to any other vertex of the type the connector points to.

4.4.3 Operations in Views

Creation

The deployment perspectives focussed on the problem of designing an appropriate interconnection of the system. Typically, the user of the framework evaluates the stakeholders, concerns, scenarios and logical view to extract requirements and constraints that determine part of the structure. Then, the user should determine what interactions between the physical and the cyber world are necessary to achieve the goals,

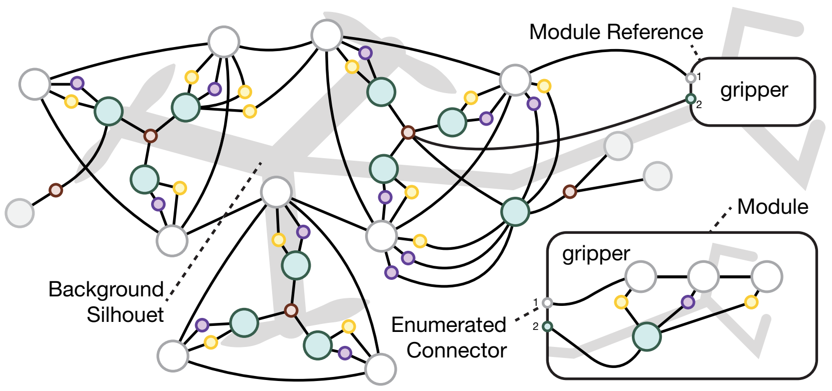


Figure 4.7 The deployment diagram ideally resembles the physical embodiment of the system of interest. This provides everyone that uses the diagram with an intuitive means to relate deployment components to the actual system. In this figure the silhouette of a drone lies beneath the other components. In this figure, the legend of figure 4.6 holds.

tasks, etc. of the system. The appropriate actuators and sensors must then be connected to the physical components and corresponding cyber-nodes. This process might involve interaction between the deployment and the process view to agree on the right structure of components and interactions.

4.5 Development Viewpoint

The previous three viewpoints treated the system from a functional, behavioral and physical perspective. The fourth viewpoint is the *Development Viewpoint* which focusses on the implementation of cyber-nodes. The cyber-nodes are responsible for realizing the behavior and functionality of the process and logical view in the cyber-nodes of the deployment view. The development view describes the structure of the components and libraries in different types of cyber-nodes.

4.5.1 Concerns and Stakeholders

The development viewpoint is helpful for addressing concerns of *architects, designers and developers* and *builders and maintainers*.

Modularity The development view describes the structure of the implementation of cyber-nodes. This viewpoint decomposes cyber-nodes into components that developers can develop individually. The developer can provide unit-tests with every module such that he can verify the functionality of the module independently of other modules.

Consistency The developer specifies the interfaces of software components in this view. Clear definition of interfaces helps in ensuring consistency between components. The developer also defines the interface between the cyber-node and the rest of the system. This interface will provide means to check for consistency in the physical view.

Reusability A development view that defines a flexible and clear cyber-node structure is reusable in other CPS as well.

Testability The development view provides features that benefit the testability of cyber-nodes. Unit-tests verify the functionality of individual software components independent of other components. The interface between the cyber-node and the *outside* enables *hardware in the loop (HIL)* of *software in the loop (SIL)* simulations.

4.5.2 Model Kinds

The view uses a layered model to describe the structure of a cyber-node. Figure 4.8 shows an example of such a model.

The bottom layer is the *interface-drivers layer* that concerns the interface of the cyber-node with other cyber-components: sensors, actuators and media. In this layer blocks represent what *driving* functionality is necessary.

The layer on top of that is the *middleware layer*. This layer contains intermediate functionality that the cyber-nodes require to realize the other layers. This includes the operating system, libraries and management functionality for managing the information exchange between drivers and MoCs.

On top of the middleware layer are the *MoC layers*. These MoC layers are responsible for providing an execution environment for the processes depicted in the process view. For example, a dataflow MoC layer provides an execution environment for dataflow models.

Creating this model requires looking at the process and deployment view to distill the different types of cyber-nodes that exist in the system. Every unique combination of interfaces (sensors, actuators, communication media) and processes yields a cyber-node type. The task of the designer is to balance between *one model for all nodes* and *a unique model for every node*. The advantage of a small number of models is the low number of cyber-node types need development. The advantage of having a large number of models is that each model has a more specific task and can be simpler to develop.

4.6 Consistency and Correspondences

Consistency is an important topic whenever more than one viewpoint or model involves in the description of a system. Although viewpoints and models aim to decouple different aspects of the system there is overlap between them. This overlap must be consistent within the AD. The *IEEE 42010 Architecture Description* standard provides the architectural concept of *correspondence rules* to help in formulating the rules that the architect must follow to achieve consistency throughout different viewpoints and models. The correspondence rules describe the relations between the different elements of the architecture description. AD elements are the most fundamental objects in the AD: stakeholders, concerns, viewpoints and their views, model kinds and

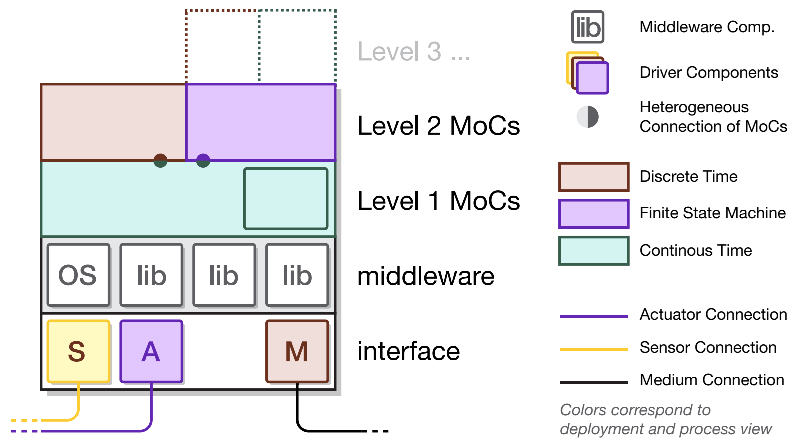


Figure 4.8 The development view describes the structure of a cyber-node. It connects the MoCs to sensors, actuators and media through interface drivers. The development view also shows what middleware is in a cyber-node.

their architecture models. Intuitively speaking, correspondence rules glue the architecture description together.

A benefit of specifying correspondence rules, next to ensuring consistency, is the guidance they provide during the process of creating and changing the system. In this section we will describe the correspondence rules between the proposed viewpoints and model in the framework. In the next section we will discuss how the architecture framework including the correspondence rules benefits the design and maintenance during the lifecycle of the system.

We also suggest to denote the consistencies between AEs through bipartite graphs: graphs of which the vertices belong to one of two sets and edges connect an element of each set. These graphs allow tracing (indirect) relations between AEs when designing by hand or Computer-Aided Design (CAD). Extending these simple graphs to include metadata could allow for more sophisticated design and validation facilities. Such an extension could be part of future research. Practical architecture descriptions will involve tens, hundreds or more AD elements and the number of relations between these elements is often much higher. These practical ADs should make use of Computer-Aided Design (CAD) techniques to allow for easy creation and modification. In the Use Case section we will discuss the usefulness of the current simple graph relations between AEs.

4.6.1 Stakeholders, Concerns and Scenarios

The stakeholders' concerns describe what properties, functionality and interactions of the system's application are relevant to them. The scenarios describe interactions and situations which are prototypical for the system. This means that scenarios should capture concerns by interactions and situations that involve these concerns. The concerns then impose constraints on the specification of scenarios.

A concern like:

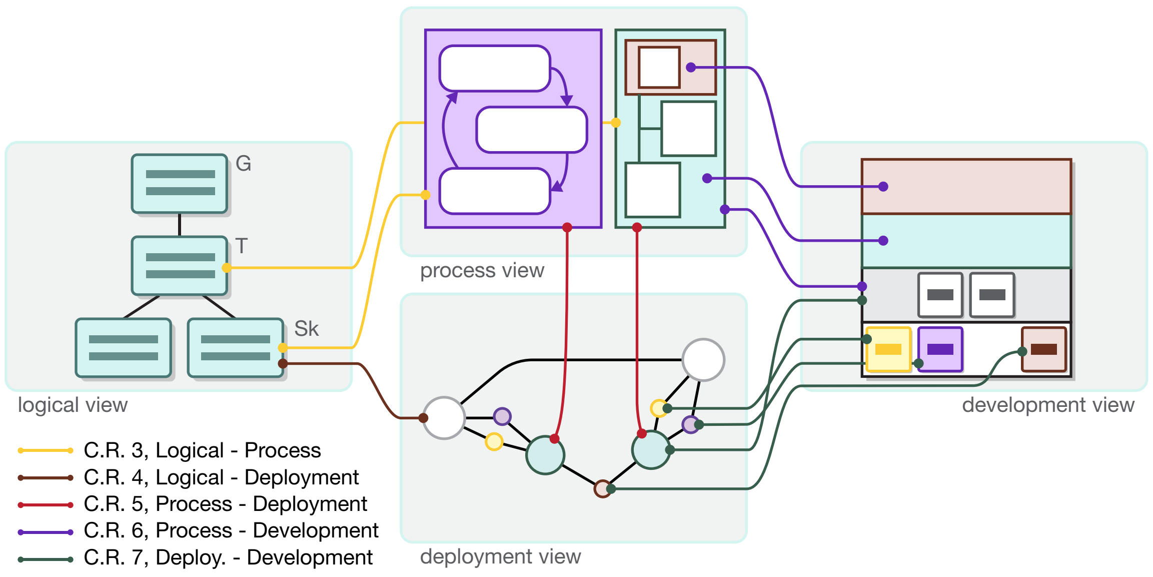

G
T
Sk
logical view
process view
deployment view
development view
C.R. 3, Logical - Process
C.R. 4, Logical - Deployment
C.R. 5, Process - Deployment
C.R. 6, Process - Development
C.R. 7, Deploy. - Development

Figure 4.9 Correspondence rule 3 to 7.

> The positioning error of the system should not exceed 1 mm.

expresses a concern that relates to the system following a reference.

This concern may translate to a scenario like this textual user story:

> The system starts in state S. The reference position is a rectangular trajectory of 1 m by 1 m. The system follows the reference position. The position error stays below 1 mm during the following of the reference.

which involves the relevant behaviour (the system following a reference) and corresponding constraints (the error is lower than 1 mm).

Note that assessing success or failure of a scenario is useful to check if the system complies to the postulated concerns. If the scenario fails then either the system or the corresponding concerns need revision. The correspondence rule is:

Correspondence Rule 1 *Concerns that involve interaction, functionality and requirements of the system should reflect in scenarios that are favourably testable through assessment. Consequently, scenarios instantiate the concepts, ideas and requirements specified by the concerns.*

The architecture description should include a list of relations (correspondences) between concerns and scenarios. A suitable model that allows expressing this relation is a bipartite graph with concerns and scenarios as vertices and their relations as edges. This is visualized in figure 4.10.

Inconsistencies

Specifying appropriate scenarios from concerns require a clear picture of what the stakeholders actually mean by their associated concerns. The risk of misinterpretation exits which may lead to inconsistency between concerns and scenarios. Asking *why* questions in discussions with stakeholders might provide information about the essence of the

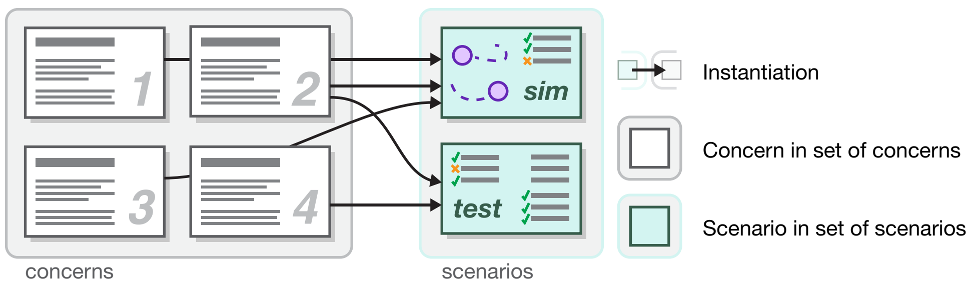


Figure 4.10 Correspondences between concerns and scenarios that satisfy the correspondence rule 1. Scenarios instantiate the concepts, requirements and ideas specified by the concerns. In this example, two scenarios exist. One is an assessable simulation, the other consists of unit tests.

stakeholders' problems and concerns and how to translate them to consistent scenarios. Reflecting early and often and involving all stakeholders therein helps in detecting inconsistencies in scenarios and concerns.

4.6.2 Scenarios and Logical Viewpoint

The logical viewpoint describes the functional components of the system in a hierarchy: goals, tasks, skills, interfaces and functions. The scenarios describe prototypical interactions and situations of the system. Together with the constraints posed by the concerns they make up the input of for the logical view. By analysing the concerns and scenarios one can derive the main functional components like goals and tasks. Then, the designer decomposes the goals and tasks in subtasks, skills, interfaces and functions up to a level of abstraction that is suitable for the goal of the architecture description.

There exists a relation between components of the logical view and the union of (parts of) scenarios and concerns. Writing down these relations allows for tracing logical elements and dependencies of these elements to scenarios and subsequently to concerns. Figure 4.11 visualizes correspondence rule 2.

Correspondence Rule 2 *Components of the logical viewpoint that relate to a (part of a) scenario should be explicitly listed in the architecture description. A suitable model that allows expressing this relation is a bipartite graph with scenarios and logical elements as vertices and their relations as edges.*

Not all elements involve directly in this correspondence rule: a *task* may relate to a concern or scenario while its *child* elements - skills, services, functions - do not. Their purpose is realizing the functionality of their *parent* elements.

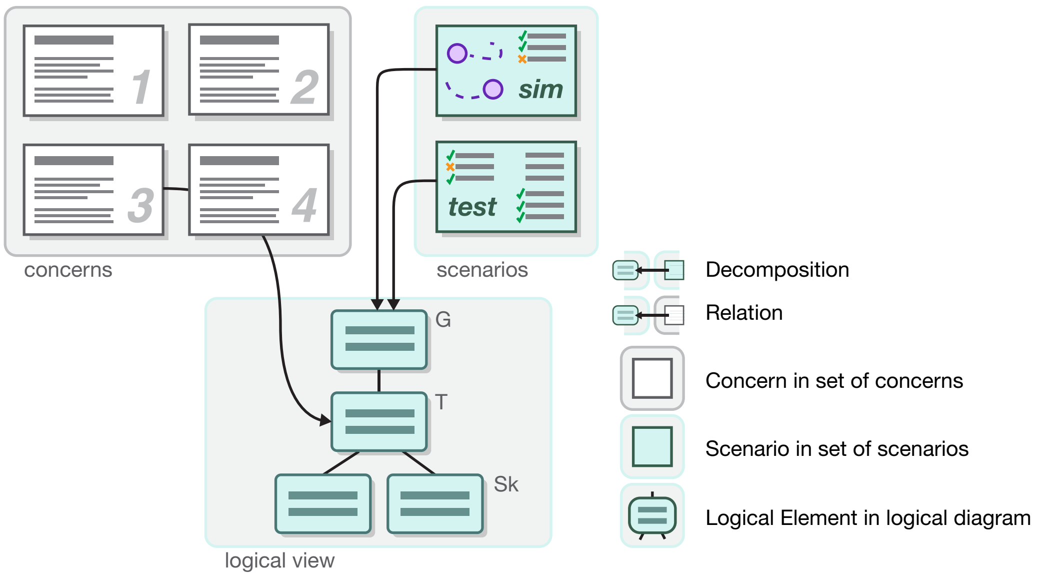

1
2
3
4
concerns
sim
test
scenarios
G
T
Sk
logical view
Decomposition
Relation
Concern in set of concerns
Scenario in set of scenarios
Logical Element in logical diagram

Figure 4.11 Correspondence rule 2 involves the relation between the union of concerns and scenarios and the elements of the logical view. Black lines are correspondences.

4.6.3 Logical and Process Viewpoint

The process viewpoint describes the system's models and their composition that gives the behaviour of the system as well as models that describe the behaviour of the environment. These models realize functionality of the logical view. The correspondence rule is:

Correspondence Rule 3 *(Sub-)modules of the process view model the behaviour of functional components of the logical view. The architecture description should make this relation explicit. A suitable model to do this is a bipartite graph with logical elements and process view (sub-)models as vertices and edges that represent their relation.*

Inconsistencies

A possible inconsistency between the process and logical views arises when an elements of the logical view are not supported by sufficient lower-level elements to achieve the intended behaviour. In this case, this inconsistency should be tracked and the missing elements of the logical view should be added in the next iteration.

4.6.4 Logical and Deployment Viewpoint

The deployment view shows how physical components of the system connect to each other and how they relate to the physical embodiment of the system and its environment. The logical view gives a first impression of what actuators, sensors, communication and cyber-nodes the system requires to realize the functional elements.

Correspondence Rule 4 *The architecture description should express the relation between logical elements and physical elements of the deployment viewpoint. A bipartite graph with logical elements and the union of cyber-nodes, actuators, sensors, external nodes and communication media ($V_c \cup V_s \cup V_a \cup V_e \cup V_m$) as vertices could serve this purpose.*

4.6.5 Process and Deployment Viewpoint

The models in the process view represent behaviour of physical components of the system and its environment. These models map to physical components of the system and the environment in the deployment view.

Correspondence Rule 5 *Models in the process view map to cyber-nodes and physical components. Connections between physical components involve variable sharing (common flow, common effort). Connections between cyber-nodes involve input-output connections (networks, data-lines). The architecture description should express this mapping. A bipartite graph could represent this mapping in which process view (sub-)modules and physical components are the vertices and the edges represent the mapping.*

Inconsistencies

Inconsistency between the process and deployment view may arise from the fact that the structure of components - as seen from the deployment view - influences the behaviour of the processes. This coupling can be insignificant, for example in input/output connections, or of essential importance, as in typical links between physical systems. If the connection between components yields changes in behaviour, this should be noted in the process view.

For example:

> Physical connection X in deployment model Y couples physical process P and Q through a common flow bond.

4.6.6 Process and Development Viewpoint

The processes of the process viewpoint belong to the domains of MoCs. Some of these processes take place on cyber-nodes. The development view describes how the cyber-nodes provide an execution environment for the processes of different MoCs. The hierarchy of composition of domains in the process view should be respected in the development view. For example, if a discrete event process is a sub-process of a finite state machine, the development view should provide an execution environment for discrete event process on top of an execution environment for the finite state machine.

Correspondence Rule 6 *The hierarchy of composition of domains in the process view should be preserved in the layered model of the development view. The architecture description should express a mapping between process domains of the process view and execution environments in the layers of the development view. A bipartite graph could serve this purpose.*

4.6.7 Deployment and Development Viewpoint

The elements of the deployment view connect cyber-nodes with physical elements and external nodes. The development view should place the *driving* functionality of these interface elements (sensors, actuators, media) in the lower layer. Each cyber-node in the deployment view corresponds to a cyber-node type model in the development view.

Correspondence Rule 7 *Each cyber-node in the deployment view should correspond to a cyber-node type model in the development view. This cyber-node type model should designate components that provide driving functionality for the interface elements (sensors, actuators, media) for all sensors, actuators and media that the cyber-node connects to in the deployment view. The architecture description should express the mapping between cyber-nodes of the deployment view and cyber-node type models of the development view. A bipartite graph could serve this purpose.*

4.6.8 Development and Scenarios Viewpoint

The development view brings together the logical, process and deployment view into a cyber-node type model. These cyber-node type models play a role in some of or all the scenarios provided in the scenarios view.

Correspondence Rule 8 *In scenarios, the cyber-node type models can be simulated, tested and validated. The scenarios that cyber-node types relates to can be backtracked through correspondence rules 3 to 7.*

4.7 Architecture Decisions and Rationale

The process of creating a system is full of making decisions. The functionality that a system supports, its performance and reliability demands and especially its realization need to be decided upon. These decisions make the backbone of a system and have significant impact on its eventual realization. The decisions themselves, their rationale, the involved stakeholders and concerns of the systems; all are essential parts of a system’s architecture and should be managed as such. The importance of decisions in a system’s architecture and the need to record them has been emphasized by Tyree and Akerman [56] and Taylor, Software Engineering and 2007 [61]. Frameworks and languages for recording decisions have been proposed [55]. Kruchten, Capilla and Dueñas advocated a specialized view for decisions in software architectures [54]. This AF chooses to adopt decisions as first-class AEs like described in IEEE-42010:2010 [24] because of their important viewpoint-supporting role.

Decisions are the fundamental entities of an architecture, while viewpoints provide the perspectives that separate the concerns of a system to make the system more comprehensible and allow for making better decisions and better systems consequently.

The user of the AF is free to select an appropriate language to capture and record decisions: Architecture Decision Records (ADRs). Available languages range from relatively simple [56, 62] to complex and feature-rich [55]. The number of decisions and the amount of related rationale in a typical project might become incomprehensible for a single person. Therefore, the chosen method should at any rate support *linking* decisions and rationale to specific elements of a system's architecture such that an interested party can easily *browse* the database of decisions and rationale.

The *Architectural Decision Records* GitHub-group provides tools for ADRs that integrate with version-controlled source-code. The Markdown ADR method [63] is a useful example.

4.8 Summary

Viewpoints provide a perspective on a system that allows focussing on an aspect without getting too much entangled by other aspects. This AF proposed five viewpoints: the scenario viewpoint, to focus on the purpose of the system; the logical viewpoint to focus on the objectives and requirements to achieve the purpose; the process viewpoint to describe the behaviour of the system; the deployment view to focus on the structure of the system and the connection between cyber and physical components; the development view to focus on the composition of cyber nodes to realize the cyber aspect of the system.

To support the viewpoints, three types of AEs are introduced: stakeholders and concerns in the previous chapter; relations, in the form of correspondence rules to describe the relation between viewpoints; architecture decisions and rationale, to make explicit the decisions made in the architecture and their rationale.

Together, these AEs provide the building blocks of an AD for CPSs. The following chapter discusses the application guidelines of these elements.

5 Application Guidelines

This chapter explains how to put the Architecture Elements (AEs) - presented in previous chapters - to use, during the life cycle of a system of interest. These guidelines constitute, together with the presented AEs, the *architectural approach*. The design and management of CPSs involves tasks like documenting, simulating, analysing and realizing. Yet, these guidelines do not dictate what exact tools the user of the framework should use for these tasks. Rather, these guidelines show and exemplify how to integrate existing tools and methods with the AF to tailor it to the way of working that the user of the framework prefers.

By decoupling the AF from the tools used, the AF's user retains the flexibility to select the best tool for the job and to use the tools they know and trust. So, as a high level *tool* the AF can and should work together with other design tools that automate tasks such as documenting, design space exploration, verification and synthesis, testing and debugging. Integrated into a workflow tailored to the specific problems that the user of the method faces, the AF becomes most powerful. Section 5.1 treats the core principles of using the framework.

5.1 Philosophy

Complexity is the main challenge of CPSs and it manifests itself in problems and difficulties throughout the life cycle. It may, for example, hide the actual goals and requirements of a system or make focussing on side issues tempting. Complexity also makes systems and their context less comprehensible to the stakeholders.

The philosophy and the core ideas of the *architectural approach* target this complexity and run as a thread through these guidelines. These core ideas guide the use of the method and help the architect in determining whether this approach suits their ideas, needs and purposes.

The following subsections describe the important complexity reducing aspects of the AF philosophy.

5.1.1 Central Role of Stakeholders and Concerns

As part of a remedy to complexity, the *architectural approach* focusses on distillation of a system's purpose before anything else. It does that by putting stakeholders and concerns central to the system, from its analysis until its reuse phase. Taking into account stakeholders and concerns is typical for architecture-centric development and standardized by IEEE 42010, but there is still much to gain from integrating these elements more deeply in the CPS life cycle than merely documenting and analysing them as independent parts of a system's design. Consequently, stakeholders and their concerns play an important role in every phase of the life cycle as they influence all decisions to be made for the system either directly or indirectly.

In the analysis phase, the user must make an attempt at identifying the stakeholders of the system and their concerns. Wherever possible, the user should distil requirements from these concerns. Together, these concerns and requirements are input to the design of scenarios and to the highest levels of the logical view. The relevant levels of the logical view in this phase are typically S_1 and S_2, goals and tasks respectively.

The following phase - design - comprises the description and working of the system's components, including tests. So, this phase is driven by the concerns, requirements and logical view derived from stakeholders in the previous phase. During this phase, stakeholders should inquire whether their perception of each other's concerns and requirements is correct.

During the later phases, the effect of stakeholders and their concerns is visible through the validating tests and use cases. Tests are the instrumental devices in which stakeholders and concerns reflect in these phases. They express the state of the system or its model during the development process and allow early detection of issues The use of testing techniques is yet another means to reduce the development complexity of systems which is discussed in the following section.

5.1.2 Iterative Design and Development

The creation of a system's architecture is not contained in a single event. This process requires the successive application of a strategy to gradually improve the architecture. If designing an architecture is a complex optimization problem in the solution space of possible architectures, then this process is comparable to finding an optimal value in an iterative way (comparable to iterative numerical optimization). We cannot find the optimal value at once because the optimization problem is too complex for us to comprehend, instead, we guess an initial architecture and start optimizing it gradually from that point on. This *iterative* characteristic is essential to the design and development process.

A rather simple design problem, for example, might have 10 design parameters, each with a value in $\mathbb{R}$, thus having a solution space in $\mathbb{R}^{10}$. Even if the parameters have limited range and their values are coarsely gridded in, say, 10 steps, the number of possible designs is enormous: 10 billion in this example. Prototyping and testing or even simulating all designs takes too much time and effort. This effect is popularly referred to as *the Curse of Dimensionality* coined by Richard Bellman [64]. A better approach would have a reduced number of parameters and to prototype and test or simulate it, then adjust some design parameters together with variants that have different design parameters close to

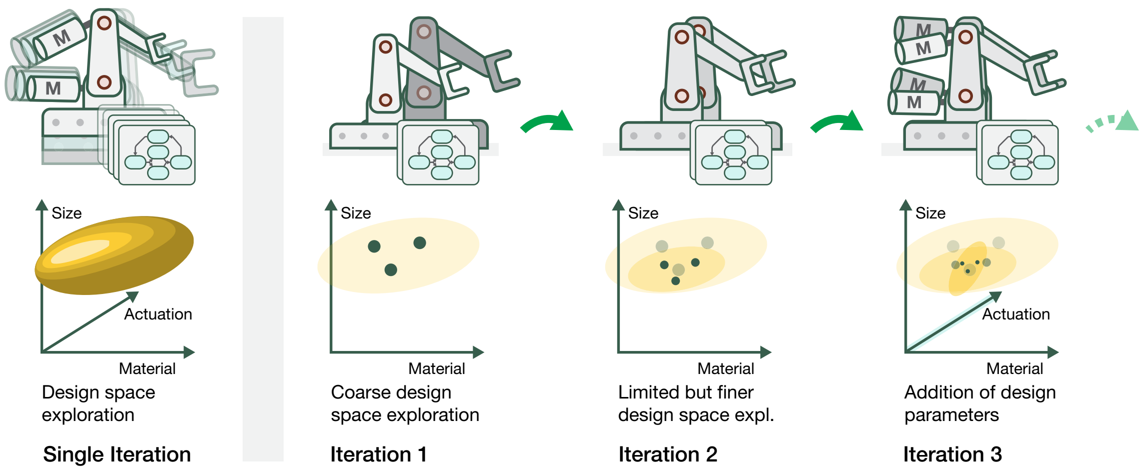


Figure 5.1 Simplified visualization of direct and iterative design space exploration. The iterative method reduces the number of designs that need evaluation which makes finding a *good* design more practical than when evaluating all possible designs at once.

the original parameters. This reduces the solution space to a fraction of the original one. The approach requires the procedure to repeat but now with the *best* design of the last iteration and with a more limited range, finer gridding and the introduction of more design dimensions. While this approach does not generally guarantee that the result is the optimal design, the alternative, evaluating all designs, is not practical.

Iterative processes are seen in many different design and development processes and have been successful in many more projects, from small to large scale. Consider an exemplary iterative design process for a mechatronic motion system. It has design parameters that influence the weight, size and stiffness of the system which affect its reliability and price. It also has less important design parameters that mainly influence the internals of the system such as motor and gear design parameters that have a minor effect on price. We start the design process by guessing one or more initial designs based on *best practices*, *back of the envelope calculations* or any other method of preference. Using these initial design choices we work out the systems architecture in all views starting with logical, then process and deployment and then development. We assess the resulting architecture by simulation and test (of scenarios) either virtually or physically. The outcome of the assessment determines what design is best. This design is the input to the next iteration which includes more finely distributed design parameters or extra design parameters that concern the internals of the system such as the motor and gear design parameters.

Figure 5.1 visualizes how this procedure differs from one in which all design parameters are concurrent.

A single iteration consists of phases that represent an effort to solve sub-problems of the architecture. Examples of these phases are *stakeholder and concern inventarisation*, *dynamical system modelling* and *field-test*. Typically, later iterations have more phases because they include more extensive models, viewpoints and tests. The difference between an iteration and a phase is that an iteration results in a complete architecture whereas phases results in solutions to sub-problems of that architecture. Figure 5.2 visualizes a typical development progress

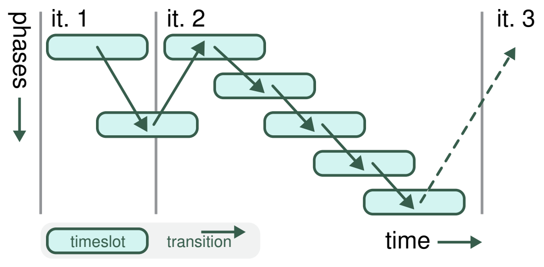


Figure 5.2 Gantt-chart of development progress that shows iterations and their phases. The first iteration consists of fewer phases than the second, this is typical for more extensive architectures.

in a Gantt-chart.

5.1.3 Testing

Testing is determining whether and how a subject complies with the given criteria. The practice of testing has acquired a central place in software and systems engineering as it allows validating systems and components already during development. This helps in early detection of deviations and errors in metrics such as behaviour, performance and reliability. Early detection is key to reducing development cost and increasing efficiency, as indicated by the (in)famous *cost of change* curve of systems [65]. Tests reduce the range of design parameters and consequently reduce complexity.

The *architectural approach* assumes that testing can play an equally important role in CPS design as it plays in software engineering. The user of the AF should consider making the following AEs testable:

Use cases Concerns and requirements constitute use cases, which act as acceptance tests.

Goals and tasks are elements of the logical view which describe expected features, tasks and behaviour that can serve as feature tests.

Skills and Services are logical view elements that describe the basic functionality and their access points. Making them testable provides interface and component tests.

Functions describe how a component or piece of software should act. Test vectors with expected outcome serve as unit tests.

A *test* or a *testable* element, typically can:

- Determine whether a (sub-)system or its (sub-)model complies with relevant concerns and requirements, automatically if possible.
- Optionally provide metrics about how much the system deviates from these goals and requirements.
- Optionally hint how changing the system influences these metrics.

An example of an automatable test is a simulation of a use case of a robot arm that interacts with an object: the simulation can determine whether the excited force, grid and speed is within range; hint on how much the metrics deviate from this range and provide information on how changing forces might impact the result.

Other tests cannot be easily automated, such as a requirement on the experience of a user interface. Such a test must then be executed manually by surveying stakeholders or a panel of users. On the other hand, these users might provide information on how much the subject deviates from the requirement, and how changing elements of the user interface might influence the requirement.

When changing a component alters the behaviour of another component, one experiences regression. Changing the mechanical characteristics of a robot might for, example change, its servo performance. Regression indicates coupling between components and though, minimizing coupling between components is good practice, it cannot be completely prevented. Regression tests focus on detecting how changes to a component change the behaviour of other components. When resolving regression issues, architectural correspondences come into play. Correspondences show how the elements of an architecture relate to each other and help in tracing regressive issues.

5.2 Analysis-phase

In the analysis-phase, *architects, designers and developers* use the architectural approach to conduct a structured analysis of the stakeholders, concerns and scenarios to derive what the logical function of the system is. They consider conceptual process views, deployment views and development views to access the feasibility of the goals and the concept system. When the *architects, designers and developers* agree on the feasibility of the system and the conceptual architecture, they use the conceptual architecture as input to the design phase.

In this phase, the architecture is in the initial stage of development. Its documentation might consist of outline documents, prototype implementations, proof of principles, lists of stakeholders and their initial concerns. Although updated and improved documents follow up these drafts throughout the development of the system of interest, archiving these initial documents is a good idea. A typical complication of complex systems' design is a system diverging from its intended purpose. Being able to keep track of the progress might help in withholding or recovering the design process from diverging.

It might be necessary to analyse some of the subsystems that the actual system-of-interest consists of. This helps the user in anticipating on, yet unknown, properties of these subsystems like for example the cost, performance, feasibility and weight. When developing, for example, a multi-agent system of drones, one has an interest in the performance of the driving mechanism of these drones. It might as well be possible that the intended system is not achievable within the resource limits available. In this phase, the user tries to detect these problems as soon as possible to reduce the cost and risk of development

Figure 5.3 shows how typical work products look like in this phase of the life cycle. In this example, the actuator circuitry, a subsystem of the system of interests, undergoes characterization. Results of the analysis phase are used in the following phases.

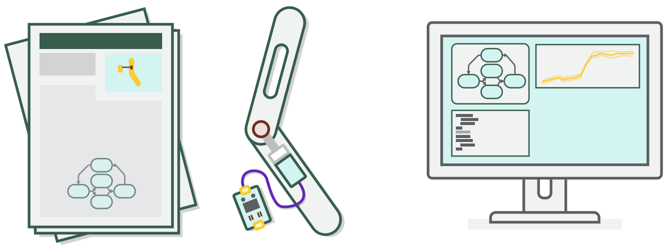


Figure 5.3 During the analysis phase, the designer tests working principles and subsystems. This enables them to use the results in the design phase and help them reduce development risk.

5.3 Design-phase

The architect starts with defining stakeholders and corresponding concerns. The architect translates the concerns to scenarios that capture the essential goal of the system. The architect decomposes the scenarios into logical components that are responsible for separate tasks to construct the logical view. The logical view is input to the construction of the process and deployment view which provide respectively insight into the behavior and embodiment of the CPS. These two views are input to the development view which involves the structure of the cyber node's realization.

This cyclic process allows an iterative way of designing CPS in which the transition from initial concept, to model, to simulation, to prototype, to the final system can take place in small steps. This reduces the risk of late discovery of inconsistencies and helps to stay focused on the actual goal of the system under consideration.

The structure of the architecture framework and the form of the architecture description it dictates allow powerful methods for validating (parts of) the architecture.

- Scenarios should be testable, preferably automated, to check if the model, simulation, prototype or finalized system complies to the application-specific concerns of the stakeholders. A scenario that *fails* encourages the developer to discover and fix problems in the system or the scenarios early.
- Unit tests can be built around individual processes to test whether they comply with their specification.
- The deployment view allows for Software in the Loop (SIL) or Hardware in the Loop (HIL) tests of parts of the CPS embodiment. The deployment view shows how a part of the embodiment is connected to another part through sensor, actuator, and communication media connections.
- The development view allows the unit-testing of middleware components that implement the functionality described in the logical view, like math-libraries, ODE-solvers, OS-schedulers, etc.

Figure 5.4 shows this development process in a visual way.

A possible method to model, simulate and validate the heterogeneous system described by the architecture description is to use co-simulation. We explained how this could be achieved with the Functional Mock-up Interface (FMI) in B. An alternative way of modelling, simulating and synthesizing (parts of) CPSs is the use of a language that supports

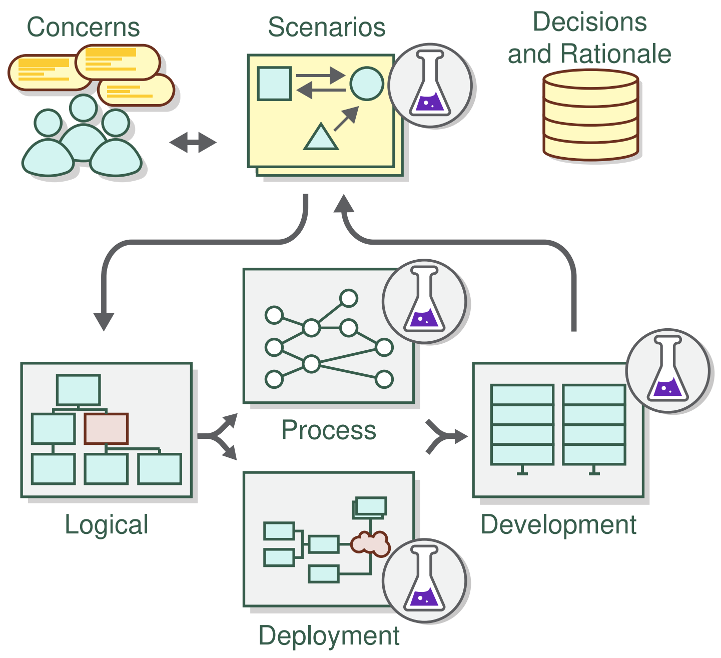


Figure 5.4 The CPS development process when using the proposed architecture. The flask indicates that the viewpoint is used for validation.

heterogeneous compositions of MoCs like SystemC. We explain this in C. SystemC is also very suitable for the creation of unit-tests in the development view because the modelling language (C++) often matches the language that the actual cyber-nodes are developed with. Figure 5.5 visualizes how the elements of the architecture can be integrated into a simulation or realization for testing. The results of this testing and simulating is then used for reporting and eventually for assessing and validating the system.

5.4 Realization-phase

During the realization-phase, the builders construct the system from the architecture description. The architecture description provides them with structures documentation about the actual intention of the system. Builders will use the deployment view to understand what components of the system interact. They can use the validation methods described in the *Design-phase* section to test if the resulting system works. If there is something wrong, they will use the correspondence relations (section 4.6) to determine what might be the cause of the failure. The architecture description also provides tools that help with communication between builders and *architects, designers and developers*.

5.5 Maintenance-phase

The maintenance-phase involves regular testing and fault recovery. Maintainers can use the same tools as builders to help them in this process. The validation methods of the *development-phase* help in checking the correct operation. The consistency rules and correspondences help in determining what might have caused a certain failure.

Issues and faults that occur during operation of the system should be tracked and linked to the corresponding elements in the architecture. When a maintainer intervenes to resolve the issue, they might help in

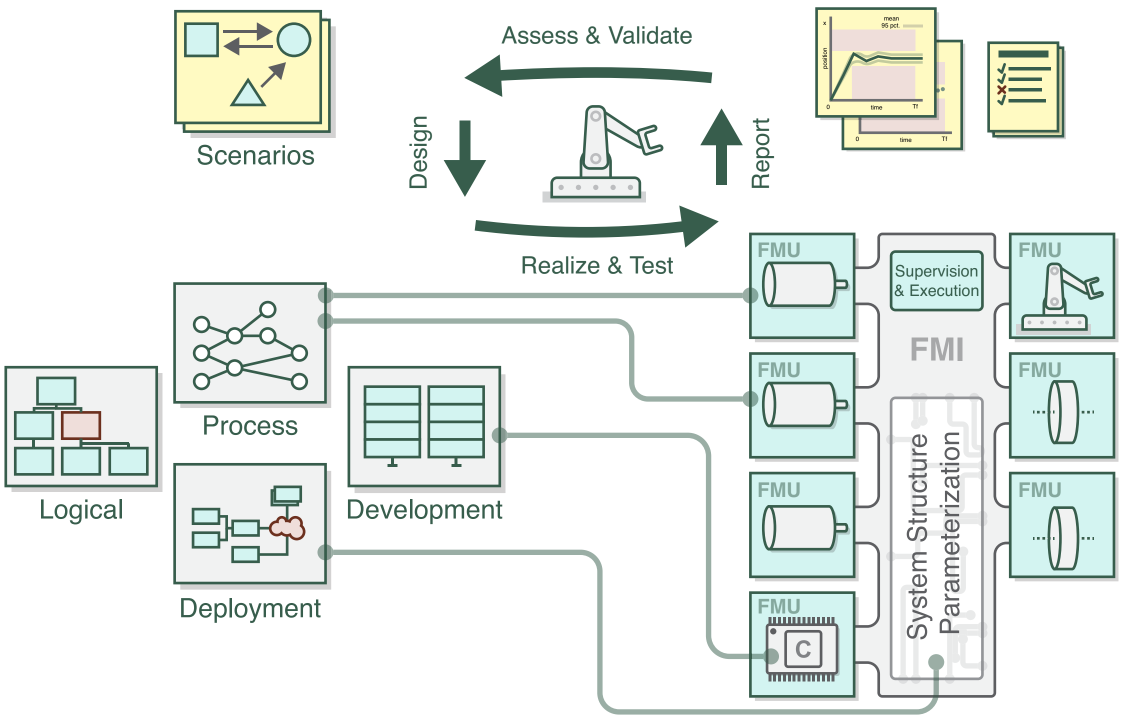


Figure 5.5 Overview of a possible structure of a development iteration. The models of the architecture are connected and combined in a test and simulation environment. In this case, through the FMI with System Structure Parameterization (SSP). Models in the process view and the development view correspond to Functional Mock-up Units (FMUs) and the deployment view corresponds to the structure of their connection: the System Structure Parameterization (SSP). The simulation environment is then used to report on, assess and validate the architecture. The results are then input to the next design step.

doing this. A suitable issue tracking system should then be present, were it a digital tracking system or an analogue piece of paper that is located with the device. The preferred method should be noted in the AD including appropriate links and directions to the tracking systems. CPSs are typically connected and sometimes intelligent enough to *detect* issues and problems. In these cases, it is highly recommended that the system itself logs issues in a tracking system such as a remote database or a log file. The resulting data can be used for resolving the issues, improving the system and tracking down related issues.

5.6 Reusing-phase

When designing a different CPS or changing an existing CPS the architectural approach motivates reusing architectural components. Changing concerns will lead to different scenarios that incorporate these changes. Functional components of the logical view can be reused to achieve overlapping concerns between systems. The relation of these functional components to deployment and processes help in determining what parts of the CPS embodiment and process composition can be reused. The relation between the process, deployment and development views helps in determining what elements of the development structure can be reused.

During this process, the validation tools that the architectural approach provides are helpful. They assess that the system achieves its intended goal and help with modifying parts of the system or parts of the scenario (and concerns) when the system does not achieve its goals.

This leads us to the following section in which we apply the architectural approach to three use cases and evaluate the process and the resulting systems.

6 Use Cases

The previous chapters introduced the elements of the AF and guidelines on how to use the framework. Yet, they provided little evidence on whether the AF is successful in its goal of improving CPS design and management.

This chapter has two objectives:

Validate whether the AF improves the design and development process of CPSs by assessing how the AF helps in addressing the CPS domain concerns throughout the design and management of the system (section 3.2).

Demonstrate the use of the framework for purposes of reference and clarification.

This chapter will treat three use cases to achieve these goals. Though, strictly speaking, use cases do not validate more than the applicability of the AF to the specific use cases, the logical fallacy of *proof by example*, they provide at least an intuition of the applicability of the framework to comparable situations. This is the most practical way to test the approach as formal proofs are beyond the scope of this report because of their complexity. The three use cases demonstrate the use of the AF as a whole, tending to the goal of *demonstration*. Table 6.1 summarizes the concerns that each use case focusses on.

The first use case involves the design of a virtual system that is able to catch an object. In this use case, we pay attention to the process of problem analysis and formulation and creation of corresponding

Table 6.1 Overview of focus concerns in each use case

Concern	**Effectiveness**	**Modularity**	**Consistency**	**Reusability**	**Extensibility**	**Testability**	**Understandability**	**Simplicity**
UC1 Ball Catcher	✓		✓			✓		✓
UC2 Powerglove				✓	✓		✓	
UC3 Production Cell	✓	✓	✓	✓	✓	✓	✓	✓

scenarios. We will then show how the scenarios and four viewpoints impact the design process of the system and how the result of an iteration forms the basis for the next iteration.

The second use case involves design improvement of an orthotic wearable device. An existing device for hand pose reconstruction and environment dynamics estimation is the basis for this device. This use cases focusses on reuse of existing devices and technologies and on *complex* physical systems.

The third use case discusses design of an industrial plant which is implemented on a scale model of this plant. The use case focusses on Hardware in the Loop (HIL) and Software in the Loop (SIL) simulations, code generation, component reusability and implementation issues.

Notation

In this section the architecture framework is applied to three use cases. The result consists of a set of stakeholders, concerns, requirements, views and scenarios: the architecture. Among the AEs that make up the architecture we provide motivation and discussion to highlight how the architecture framework applies to the problem and what are its advantages and disadvantages. Blocks that are marked-up corresponding to their type will contain the different types of AEs. Every architecture description benefits from a clear textual and visual style to guide its *reader*. Though the style used throughout this chapter may be taken as a starting point, we will not impose any style requirements on the user of the framework to allow them to choose a style that suits the intended audience best. No single best style exists because the most suitable style will depend heavily on the form of the architecture description.

Concerns **must have** a system-wide unique identification number and **must have** one or more stakeholders. Concerns feature a finger-up icon, indicating their stakeholder's need for attention and have an identification starting with a C and post-fixed with ex in case of explanatory examples like the three listed below:

C0.1ex - Organizational Concerns *(Example)*
Stakeholder(s): *Architect*
This block describes the architect's concern which might be rather general (reliability, organization) or more specific (the system must conform ISO42010).

Scenarios describe situations that test a system. They include rules that let the system pass or fail, comparable to expressions in programming languages. They **must have** a system-wide unique identification number and feature a book icon, referring to their often story-like occurrence.

S0.1ex - Faulty Powersupply *(Example)*

- This scenario describes a situation that involves a faulty powersupply
- During normal operation the voltage drops to zero with a chance

of 0.1% per minute.

- The system should shutdown to pass.
- The system fails if the shutdown takes longer than 4 seconds.

Requirements **must have** a system-wide unique identification number and **must** relate to architecture elements and **may** relate to architecture decisions that explain and motivate the requirement. They feature a checkbox icon, referring to their need to be *checked*.

R0.1ex - Requirements according to ISO42010 *(Example)*
Concern(s): Reliability Concern
Logical Element(s): Motion Control Service
Decision(s): ADR-1

- The bandwidth of the motion control must be $100\,\mathrm{rad\,s^{-1}}$
- The overshoot of the motion control must be less than $5\,\%$

Comments that discuss the architecture framework in the context of a use case are provided. These comments exist to discuss the application of the framework and would be absent in any independent AD.

What stakeholders say and what they want *Example*
When determining the concerns of stakeholders, it is important to realize the discrepancy between what a stakeholder says and what one wants. For example, a stakeholder might pose a concern that the system should be real-time while, essentially, the stakeholder wants a system that is reliable and safe.

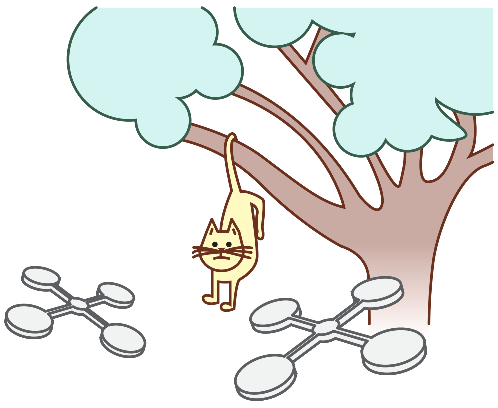


Figure 6.1 Pet Catch system will involve a pet that is at height and a number of existing surveillance drones.

6.1 Cooperative Pet Catcher

The fire department experiences busy times due to dryness and extended periods of heat that cause fires. At the same time, the usual number of pets climbs up trees to become too scared to get down again. The fire department has a number of surveillance drones available that might be helpful in these situations. The goal of this system is to catch pets that are to fall from trees and other heights. Figure 6.1 shows an impression of this situation.

This use case focusses on **effectiveness**, **consistency**, **testability** and **simplicity** concerns. By demonstrating how the framework helps in translating stakeholder's concerns to a suitable architecture, the effectiveness is evaluated. Consistency is discussed in the context of creating views and the correspondence rules that bind their elements. The testability is assessed by evaluating how the framework supports testing of parts of the architecture. Eventually, this difficulty of applying the framework is reviewed to assess how the simplicity concern is addressed in this situation.

To study the feasibility of this system and develop a concept, the AF will be applied to this problem. The starting point is to inventory the relevant stakeholders and their concerns.

6.1.1 Stakeholders and Concerns

A situation like this may sound strange at first and raise many questions but dealing with stakeholders will involve bridging a perception and language gap. Stakeholder and concerns analysis helps in distilling this purpose of a system from its stakeholders.

The framework provides us with a starting point of three typical stakeholders: first, those that design the system (architects, designers, developers); second, those that construct the system (builders, maintainers) and third, those that use the system. In this use case, we use the framework to design and construct a system to solve a problem we have ourselves: we, a group of persons, embody the three typical stakeholder groups. An additional stakeholder group is involved: the readers of this document, interested in the motivation, usage, advantages and downsides of the architecture framework.

The stakeholder-concern mapping of the system under design is based upon the starting point provided by the architecture framework in table 3.1 because the system's stakeholders include the default three. Addition of the additional stakeholder group, readers, and two additional problem-specific concerns tailor the stakeholder-concern mapping to the use-case.

C1.1 Catch Pet Safely
Stakeholder(s): End-user
The user wants the system to catch the pet without it being hurt.

Next to the end-user, the reader poses an educational concern: they want to understand how the architecture framework influences system design. The stakeholder group of readers has an interest in the demonstrative purpose of the system.

C1.2 Demonstrative Purpose
Stakeholder(s): Reader
The system and its design must demonstrate the typical usage of the architecture framework, including advantages and disadvantages.

6.1.2 Scenarios

The next step is to construct a scenario that captures this concern. In the form of a story:

S1.1 - Cat falling from tree
A cat is in a tree, 3 to 8 meters above the ground. It is too afraid to come down and too tired to stay so the cat jumps down.

The initial velocity of the cat is a 3-dimensional vector with magnitude less than $2\,\mathrm{m\,s^{-1}}$. The system predicts the location of impact and catches the cat.

- ✓ The system should catch the cat
- ✓ The cat must not be exposed to unsafe accelerations
- ✓ After catching, the cat must be brought near the ground

Figure 6.2 shows a simulation of this scenario in a 20-Sim 3D animation window. The scenarios provide the designer with tools for both architecting and testing the system.

- Scenarios are input to the logical view construction and help the architects, designers and developers to reason about the required functionality of the system.
- The designer can use the scenarios to assess whether the system addresses the concerns of the stakeholders, both manually and automatically.

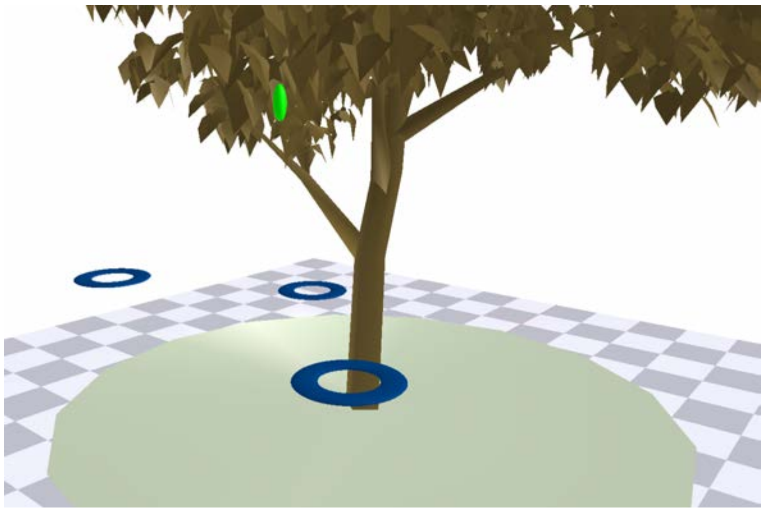


Figure 6.2 Three-dimensional visualization of the scenario in 20-Sim. Such a visualization helps the stakeholders in aligning their expectations and concerns and the designers and builders of the system in assessing its correctness of operation.

6.1.3 Logical View

The scenarios help in determining what the main goal of the system is. The essence of the application is to *save pet*. This goal is then decomposed into two enabling tasks: to actually catch the pet and to move it to a safe location. These tasks require three skills: *object trajectory prediction*, *collaborative movement* and *collaborative localization*.

- *Object trajectory prediction*: Requires a service that can predict dynamics of objects. The involved differential equations need the functionality of solver algorithms.
- *Collaborative movement*: Requires services for configuration planning, position control, broadcast communication and current state estimation. Configuration planning relies on functionality for determining the appropriate location; position control relies on force control functionality; broadcast communication requires transceiver functionality and state estimation can reuse the differential equation solver algorithm functionality used in the *object trajectory prediction* task. To estimate the state of an element we choose to incorporate acceleration and gyration sensing functionality.
- *Collaborative localization*: Requires services for state estimation, broadcast communication, object localization and distance measurement. This skill can share state estimation and broadcast communication with the *collaborative movement* skill. To enable object localization, some localize functionality is used. Trilateration, triangulation, computer vision or another technique could be used but no choice has to be made: a placeholder *localize technique* is inserted in the logical view. Also measuring distance requires functionality that is not fully specified at this early phase of development.

The choice for *collaboration* of multiple *catchers* is at this moment a

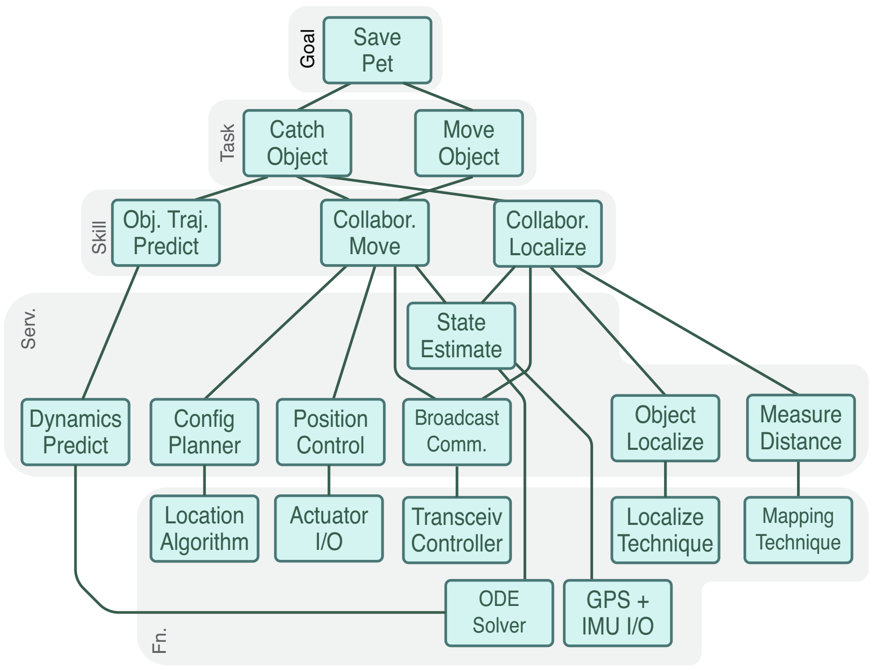


Figure 6.3 Hierarchical diagram that relates goals, tasks, skills, services and functions. The diagram is part of the logical view of the system.

result of an architecture decision.

Services provide functionality to in turn enable the skills in an application independent way. That means that through the skill knows about the available services, the service is not designed specifically for the skill. These services are access-points to system functionality like algorithms, libraries, controllers, solvers, sensors and actuators. Because the services are not specific to the corresponding skills, they can more easily be used to enable more than one skill. Also, reusing components is easier. Figure 6.3 shows the logical view of the system. Now that the goals, tasks, skills, services and functions are defined, the corresponding behaviour in the process view and their physical embodiment in the deployment view must be designed.

In early iterations, it might not be evident which functionality is appropriate. The concerned functionality can then be modelled in a more general way, like in Figure 6.3: it is unclear what function is needed to support the 'measure distance' service, so it is substituted by a general placeholder. Choosing a specific implementation can then be postponed to a later moment in the development of the system when more constraints are in place and context is better known.

However, a significant downside of this is the possibility that the functionality is not possible within the posed constraints. Such an issue might then lead to a significant increase in costs with respect to earlier detection of the impossibility. To avoid situations like these, the user should provide rationale with its decisions, based on experience, literature or an analysis.

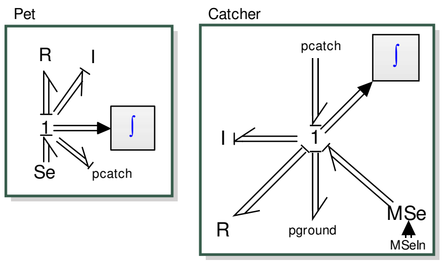


Figure 6.4 Process view of the physical processes of the CPS and its environment. A three-dimensional one-junction with gravity, inertia and resistance models the object (pet). A three-dimensional one-junctions with gravity, inertia, resistance and a modulated effort-source model the catcher.

6.1.4 Process View

The process view describes the behaviour of the physical and the cyber components. The process view of the physical parts of the system describes the behaviour of the catchers and the object to catch (the pet). It can be identified from the scenarios: there are *catchers* and there is an *object to catch*. This behaviour can be modelled by bond-graphs and in this use case, 20-Sim was used as modelling tool to do this. Bond-graphs are a graphical representation and a dynamical model at the same time which is beneficial for documentation and analysis. Figure 6.4 shows the model of the physical process.

The processes view of the cyber parts of the system describes the behaviour of the cyber-nodes. The model is a composition of MoCs that provides the functionality of the logical view.

Creation of the process view is typically done by modelling the processes needed to achieve the goal, then the tasks, and so on. The main goal has two tasks that are about coordination and movement of the catchers. These two tasks could be implemented in a single process.

The *object trajectory prediction skill* is used to help the catcher extrapolate the perceived state of collaborating catchers and the object to catch. To achieve the *catch object* and *move object* tasks, the three skills need to collaborate in a process. Four parallel processes are identified by grouping the services of the logical view:

Coordination A process takes care of analysing the available data to estimate the system state and to plan future actions. This process can use a FSM MoC. Internally, it uses dynamics predication and state estimation services. These regular processes can be modelled in a DT MoC.

Positioning Takes care of controlling the position of the drone based on sensor measurements. Controllers need regular execution and a SDF MoC is a relatively simple model that allows this.

Information Exchange Takes care of regularly transmitting measurements and receiving those of other catchers. A FSM is initially chosen as MoC.

Communication Is responsible for low-level communication, such as carrier sense and queueing. A detailed design of this process

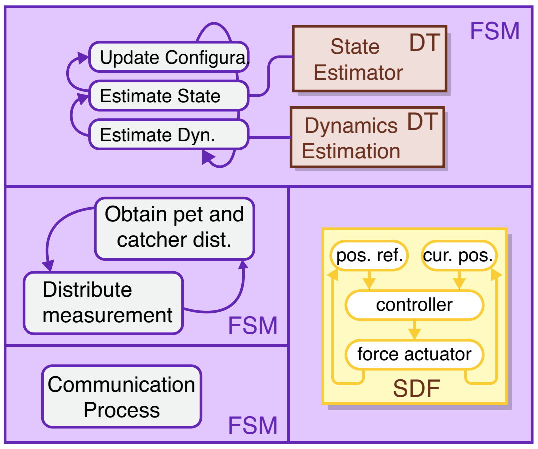


Figure 6.5 Process view, model of the processes that realize the functionality of the logic view.

is not necessary in this early phase as communication can be mimicked in tests.

The selection of MoCs is a design challenge that requires insight into the problem and the possible solution to solve. In the end, the core difficulty of creating an architecture remains to make the right decisions. The framework does not make decisions but rather it helps its user in making better ones. Determining what is the *right* decision might require a process of trial and error which fits fine within the framework. An important decision should be recorded in the architecture's decision repository as a ADR with its corresponding motivation and rationale.

Figure 6.5 shows the resulting process view of this design iteration.

6.1.5 Deployment View

The deployment view captures the relation between components of the CPS and the physical environment.

We choose to equip each mechanical element with a cyber-node, a 2D planar force actuator, a planar accelerometer and gyroscope for determining the position and an ultrasonic distance sensor that measures the distance of the element to the ball.

The diagram in Figure 6.6 shows the relation between cyber-node, medium, sensors and actuators for a single mechanical element as supposed in the scenario. We can construct the full deployment graph by repeating the shown vertices and edges for every mechanical element.

6.1.6 Development View

The combination of the process and the deployment view specifies the types of cyber-nodes that exist in the system by a mapping of MoCs

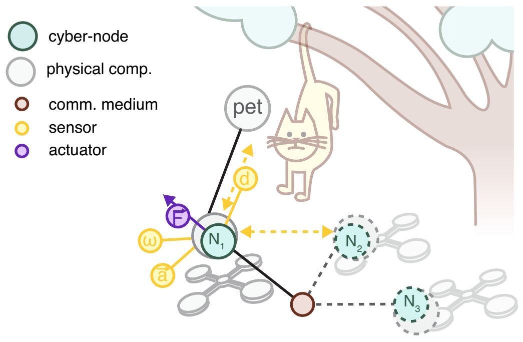


Figure 6.6 Deployment view, model of deployment for a single element and cyber-node.

to cyber-nodes. The number of types of cyber-nodes should be as low as practically possible to ease development, testing and updating. Typical valid reasons to use different types of cyber-nodes are that the performance requirements differ between nodes such that using high-performance hardware and software on every node is unnecessary and expensive.

In this system, there is one cyber-node type since the catchers are functionally equal. We can determine the communication medium and connected sensors and actuators for this cyber-node type from the deployment view: An IMU, force actuators and a communication interface and distance sensor of which the exact implementations are not yet decided upon in this iteration. These devices are the interface of the catcher's cyber-node type with the physical world (through sensors and actuators) and the rest of the cyber world (through communication). This interface is on the lowest level of the development view.

The middleware layer contains libraries and supporting functionality, based on the required functionality and the processes. In this case, an ODE solver and trilateration algorithms are the supporting components in this layer. Because the process view has concurrent processes, a mechanism that manages this concurrency is necessary. Simulation environments take the responsibility for such a mechanism but when implementing the cyber-nodes on hardware, a scheduler component - typically part of an Real-Time Operating System (RTOS) - is necessary.

The MoC levels follow from the process view: the base level is an FSM and the second level contains SDF and DT processes. Note that it is possible that different cyber-node types realize different parts of the cyber part of the process view. In this use case, there is just one cyber-node type - that of the catcher - so all processes are present in the development view.

Figure 6.7 shows the resulting development view. This view shows that there are two MoC interfaces: FSM-DT and FSM-SDF. The appropriate way to realize this interface depends on the situation. In this first development iteration, we suggest using methods provided by a heterogeneous simulation environment, such as Ptolemy II or ForSyDe.

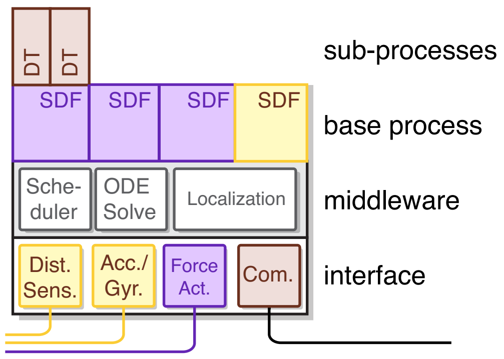


Figure 6.7 Development view, model of the software structure of the system

6.1.7 Concluding the Iteration

We now have everything we need to complete the development iteration by putting the resulting views together and assessing the scenarios. This integration of views and realization of a result (simulation, prototype, product, report) will involve effort from all stakeholders of the system. Though early integration can be difficult, integrating all views only at the end is doomed to fail: views might have diverged and issues have not been discovered and resolved to lead to incompatibilities that are far more complex than those detected earlier in the process.

The development view, process view and deployment view together can form an executable system. The designer can simulate the system in a simulation environment, use code-generation or coding by hand to build the system on actual hardware or a bit of both with hardware-in-the-loop and software-in-the-loop simulations. In this first development cycle, simulations are often the way to go. Figure 6.8 shows the architecture in Ptolemy II, the similarity between this model and the deployment view is noteworthy. The physical process view, shown in Figure 6.4 is reusable in this model. The Functional Mock-up Interface version 2.0 (FMU 2.0) allowed compilation of the original 20-Sim model for incorporation in the Ptolemy II model. Appendix B gives more info on this topic.

Executing the simulation will allow assessing the architecture to validate the result and discover issues. A new development iteration can be started, or in some cases, the development might be ceased. If the scenario failed, either the scenario or the architecture needs revision. Checking the scenario is important to see whether *failure* or *success* indeed indicates that the concerns are respectively violated or complied with.

6.1.8 Further Iterations

The number of iterations that the user can apply the framework is not limited. The framework's user chooses when to stop iterating and

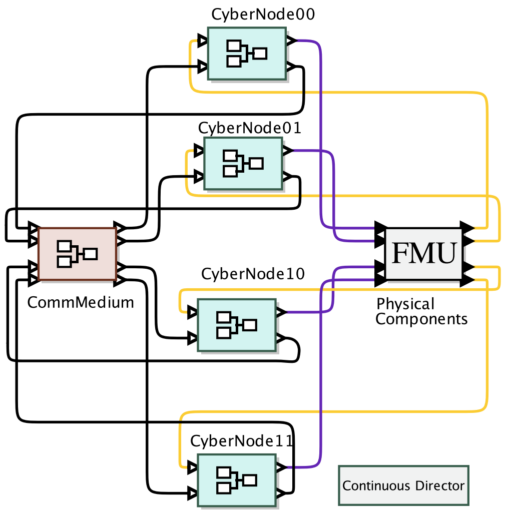


Figure 6.8 Heterogeneous Simulation in Ptolemy II with co-simulated 20-Sim model. The colouring of the diagram reflects the component types of the deployment view: teal cyber-nodes, brown communication medium, grey physical components, yellow sensors, purple actuators.

for what reason, whether it is time, money, satisfactory results or a combination of these and other reasons. Most systems that involve different parties and deal with a high number of concerns are never eternally finished because the environment, stakeholders and concerns may change in unexpected ways, requiring the system to adapt.

In later iterations, more realistic simulations could be created to eventually realize a physical system. HIL and SIL technologies will allow us to make this transition gradually.

Wrapping Up

The advantages of the architectural approach in this use case are:

- The framework provides a thread that guides us through the process like a step-by-step guide.
- It helps in translating vague requirements and concern to tangible goals, task, etc.
- The set of views provided relatively independent perspectives on the system.
- The system and scenarios are automatically testable.

The main disadvantages of this use case are:

- The selection of appropriate MoCs, functions and implementations remains difficult.
- A top-down approach risks the possibility that implementation details make certain choices impossible.

The **effectiveness** of the system's architecture benefits from the structured analysis of the initial requirements and concerns to goals and tasks. The corresponding scenarios allow assessing the system's effectiveness during its development.

The architecture is described using largely independent views, but to fit together they must be **consistent**. By providing correspondence rules and means for testing the resulting architecture, this consistency can be checked. This, however, does not guarantee the consistency and when integrating the views to realize the system this can lead to integration problems. By keeping iterations relatively small, inconsistencies are kept correspondingly small.

The architecture views of the system are to some degree individually **testable**. Through the scenario view, the framework provides an integration test for the system. The guidelines for applying the framework help in creating a simulation architecture, in this case in Ptolemy II and 20-Sim.

The framework forces through its top-down approach a goal-oriented way of working: try to get a simple prototype working as soon as possible. The **simplicity** of the system might benefit from this as long as implementation issues are detected and resolved in time.

Although this use case of applying the *architectural approach* is no formal proof, it portrays its characteristic advantages and disadvantages. The discussed principles are applicable to other CPS design problems as well.

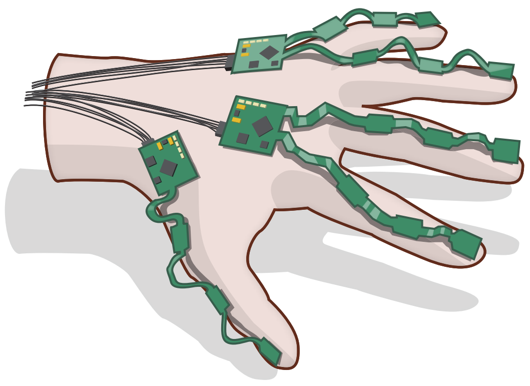


Figure 6.9 Illustration of the PowerGlove with three devices of four (for 1 finger) or seven (for 2 fingers) 9-DOF IMUs attached for a total of 162 measured DOFs

6.2 PowerGlove Use Case, Architecture Recovery

The starting point of this use case is an existing system (Figure 6.9), developed in previous work, to estimate the hand pose [66] and energy expenditure of a person [67]. This device bears the name *PowerGlove*, it utilizes Hand-pose Reconstruction (HPR) algorithms to estimate the pose of the hand from time-series acquired from IMUs.

In this use case, an architecture is recovered from an existing system. A benefit of architecture recovery is that it helps in making a system better **understandable**. This architecture can then be used to analyse and eventually extend the system in an effort to demonstrate the **extensibility** concern. The use case shows by example how the AF addresses **reusability** concerns.

The PowerGlove is developed independently of the AF and no actively maintained AD is currently available, so the first step is to *obtain* one. This is the process of *architecture recovery*, creating an AD from a system without a known AD. The recovery follows the familiar steps of the AF, possibly iterative to fine-tune the architecture to sufficient detail. A complete recovered architecture may be used for documentation, reference and as a starting point for extending the system with extra functionality.

First, the stakeholders and concerns are re-evaluated and the uses of the system are described in the form of scenarios. The logical view is built by decomposing the uses into goals and requirements based on the scenarios. The process and deployment views are built by assigning the structure and behaviour of the system to respective nodes and processes, then assigning these elements to their respective logical elements. Then, the deployment view is constructed by evaluating the layered structure of the cyber nodes: drivers, supporting libraries, models of computation used, then assigning these elements to their corresponding processes and nodes.

Decisions that significantly influence the system, such as should be tracked and linked to the corresponding AE in the form of Architecture Decisions (ADcs). ADcs are an ideal place to store pieces of information recovered from the original system, such as notes, literature references, code comments and models.

6.2.1 Stakeholders and Concerns

The original system stakeholders are generally recoverable from available documentation, research papers, notes and work logs.

The architecture framework specifies three standard groups of stakeholders. In this specific system, the *end-user* stakeholder is replaced by two more system specific stakeholders:

Clinicians use the system to assess the functioning and performance of the hand.

Patients wear the system to have their hand assessed.

And a third stakeholder group is added:

Researchers use the system to test and evaluate new technologies and algorithms.

The resulting five stakeholder groups cover the most important groups that have unique interests in the PowerGlove system.

The system-specific concerns of the PowerGlove are, based on documentation and papers:

Data Availability All measured DOFs of all fingers must be available for reference and analysis

Pose Reconstruction The pose of the hand must be reconstructed from the data

Assessable The pose of the data must be assessable, both visually and using clinical performance metrics

6.2.2 Scenarios

To further recover the architecture, typical usage scenarios are distilled from the stakeholders, concerns and existing documentation [66, 68] of the system.

The most typical scenarios are those in which the glove is used to record data and those in which the device is used to reconstruct and assess the hand and its functioning.

S2.1 - Data Acquisition and Recording

- This scenario describes a situation in which a person wears the device.
- The device is appropriately enabled (connected, powered).
- The operator (researcher, patient or clinician) *starts* and *stops* the recording.
- ✓ The system must record the movement data of the hand of the person and present this data for further processing or storage.

S2.2 - Hand Pose Reconstruction and Analysis

- The conditions equal those of S2.1
- ✓ The system must reconstruct the hand pose and assess relevant clinical performance metrics and present these results to display or to allow further processing.

Another scenario describes how the device should influence the user.

S2.3 - Using the Device as a Wearer

- The user wears the device in a clinical setting during performance assessment.
- ✓ The user spells the sign language alphabet with the hand wearing the device. Throughout the process, the system must not impede the movement of the wearer. Whether the movement was impeded or not is assessed by directly asking the wearer.

The third scenario describes an interaction between the system and the wearer and a condition that indicates whether the system functions correctly. The current way of assessing the result is by asking the wearer a question, indicating that a physical device must be present. Some scenarios are not automatically assessable, like S2.3. Where possible, however, try to design scenarios that can be automatically assessed or at least simulated because this enables a range of tools that can improve the design process.

The resulting scenarios can now be discussed by the stakeholders to align their expectations.

6.2.3 Logical View

From the listed scenarios and existing documentation, the logical view can be constructed.

The core goal of the system follows from the previously described functionality and concerns: enable the evaluation of the hand pose. This functionality is of a high-level and requires supporting system tasks to succeed. These tasks are: estimating the hand pose of the wearer; assessing the performance of the user; exposing the results to the environment.

Tasks, in turn, rely on basic functional skills of the system. Skills provide a means to transition from high-level tasks to actual system configuration. Skills are supported by services, which provide the interface of the system to potential subsystems and components. The tasks of the PowerGlove are decomposed into tasks, skills and finally functions and summarized in Figure 6.10. Note that this figure is just a diagram and

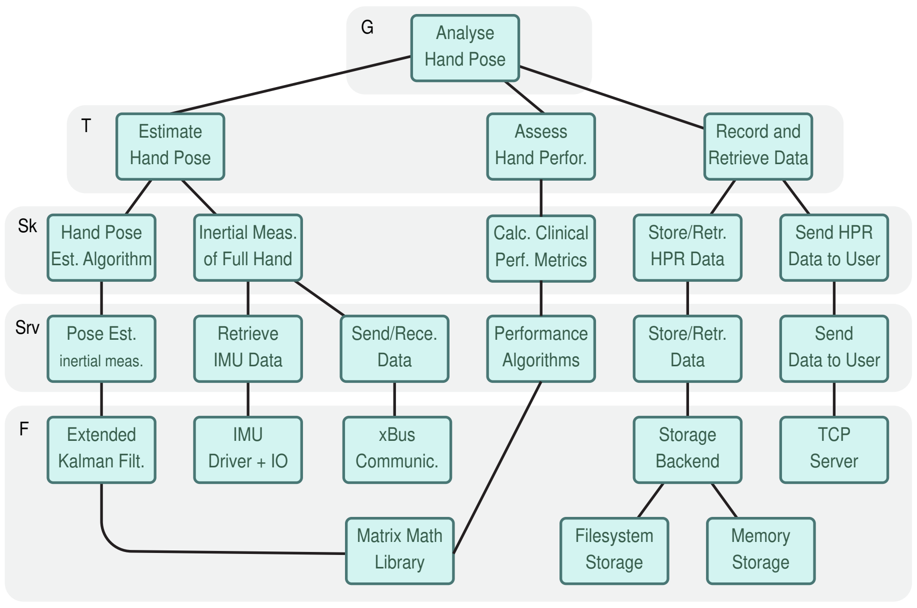


Figure 6.10 Diagram of the logical view of the PowerGlove. Higher-level elements coordinate their sub-elements to achieve their goal. The *estimate hand pose* task, for example, relies on skills for *retrieving mechanical IMU data* and *mechanism pose estimation*

that the actual way of representing the view is free for the user of the framework to choose. For example, by storing the logical view textually in the code base of the system.

At this level, requirements can be incorporated in the view too. In this use case, that comes down to recovering the original requirements and their motivation from the system.

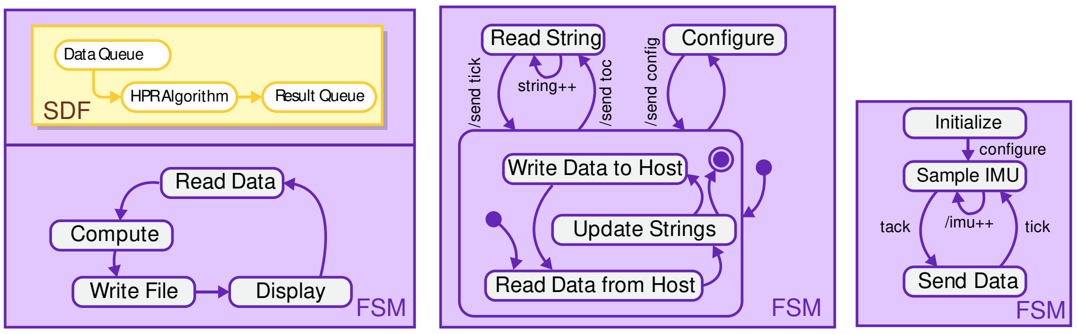
A requirement that plays an important role is:

R2.1 - Sample rate of Hand Pose and Inertial Data
Affects: Estimate Hand Pose Task, Expose Results Task
Related Decision(s): ADR-PG-0001-decision-on-sample-rates

- The sample rate of gyroscopic data must be at least 200 Hz
- The sample rate of acceleration data must be at least 100 Hz
- The update rate of hand-pose reconstruction must be at least 20 Hz online ‘real time’ and 50 Hz offline post-processed.

The ADR includes references to appropriate rationale and research, the considered alternatives and the motivation for the decision made. The documentation and notation approach that suits the system under consideration best also remains free to choose to the user of the framework.

An independent AD typically includes more decisions and rationale but the number of ADRs tracked in this use case is kept low for reasons of brevity. The decisions are listed in Appendix D.2.



Recording and HPR process   Data acquistion and configuration   Reading IMU

Figure 6.11 Process view of the PowerGlove, based on the existing system.

6.2.4 Process View

The process view describes the composition of processes that are responsible for the behaviour of the system. This composition should support the functional elements in the logical view. The process view will be based on the processes in the existing PowerGlove and will in this iteration only incorporate the cyber-aspect of the system: those responsible for processing the data.

The three main processes in the system are: the HPR and data recording process, this runs typically on a powerful computer; the data acquisition and configuration process, responsible for acquiring IMU data, distributing a common clock signal and preparing the data for transmission to another device (the host); the IMU read-out process.

The process view is visualized in Figure 6.11.

During architecture recovery, it is possible that the architecture of the system of interest has (minor) inconsistencies. For example, some functionality might not be realized or requirements might not be fulfilled. Sometimes, this analysis gives insight in bugs and issues in the system or what aspects of the system can be improved.

6.2.5 Deployment View

The original PowerGlove uses 3 sets of IMUs per hand device to retrieve sensor data from 7 or 4 IMUs and a coordinating device to retrieve data from the 3 sets to sending it to the receiving device: typically a computer.

Each set of IMUs has a dedicated microcontroller, the coordinating device has another microprocessor. This makes a total of 5 cyber-nodes, including the host computer. The full deployment view is shown in figure 6.12.

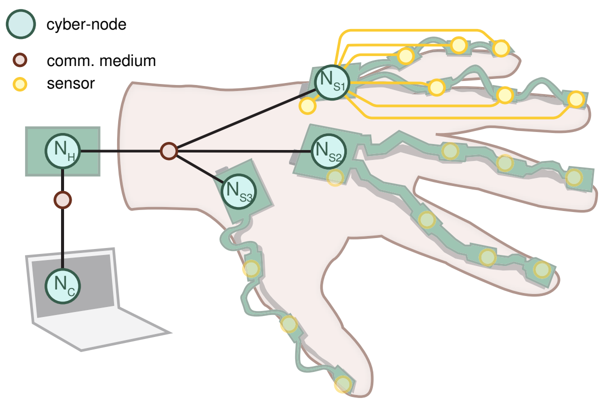


Figure 6.12 Diagram of the deployment view of the PowerGlove. The nodes N_{S2}, N_{S3} have their sensors omitted but do connect to the same sensors as N_{S1}.

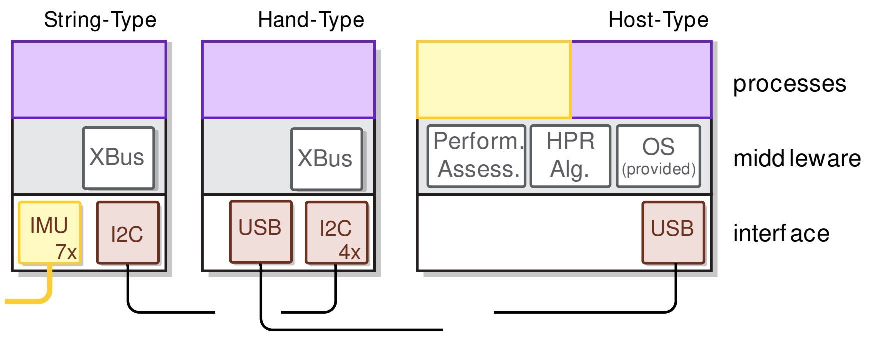


Figure 6.13 Diagrams of the development view of the PowerGlove. Three types of cyber-nodes are used. The String-Type and Hand-Type on the PowerGlove itself and the Host-Type on the computer.

6.2.6 Development View

Comparable to the way that the development view of the cooperative pet catcher is constructed, the designer can create a development view for the cyber-node types of the PowerGlove. The main cyber-node types in the original PowerGlove are the *computer* type which represents the device used for visualization (N_C), the *hand* type, which represents the cyber-node N_H and the *string* type which represents the cyber-nodes that connect to the sensors on the fingers (N_Sn).

The required sensor, actuator and medium drivers follow from the deployment view (Figure 6.12). The required MoC layers follow from the process view. Figure 6.13 shows the development diagrams of the three types of cyber-nodes.

The middleware that the cyber-node types require depends on the MoCs, algorithms, sensors and actuators used. A cyber-node that uses a single main FSM does in theory not need any scheduling or Operating System (OS) and is implementable in C or any other embedded programming language. However, this is a typical example of a *leaky abstraction*: the model simplifies something through abstractions but the abstraction is affected by the underlying implementation. In this case, interrupts, background data-transfers and communication will interrupt the execution of the FSM and this may cause it to behave differently. In this case, the decision (`ADR-PG-0002`) is to accept the *leak* in the abstraction and handle and document it appropriately.

- The discrepancy is taken care of by ensuring that interrupts have no significant impact on the behaviour of the FSM.
- The shortcoming of the FSM process model is document in the AD.

Models and abstractions try to approximate the underlying phenomenon. Eventually, they diverge from the phenomenon they represent. This can be dangerous and misleading to those involved in the model. When the divergence between model and reality has significant consequences, there are typically two things one can do: either use a more realistic or specialized model or accept the divergence and handle it as well as possible: document and handle the consequence appropriately.

The computer-type cyber-node does require the concurrent execution of another MoC. The computer has an OS available so this can take care (abstract away the implementation details) of concurrency. Existing OS facilities are used to support communication between processes.

6.2.7 Concluding the Iteration

This iteration demonstrated the creation of an AD from an existing system.

The next step can be to model and simulate the new *hand* cyber-node type and run a software-in-the-loop experiment with the existing PowerGlove to start a new development iteration. In this new iteration, the views may be detailed out or expressed in an executable graphical model such as in Ptolemy II or Simulink.

To improve and automate scenarios, a welcome addition would be to add the behaviour of the physical components of the system to the process view. This includes the dynamics of the hand, the user and the objects that it interacts with. Reusing existing biomechanical models of the hand - like that of Peerdeman et al. [69] - is a solution. In general, one should consider using, adjusting or extending existing models of environments and systems to avoid duplicate work.

The proposed scenarios cannot be simulated in this iteration because there is no model created for the ‘environment’ of the system (the wearing hand). Situations like these, in which complex dynamics are involved, limit the possibilities of simulation and automatic testing.

The following section will sketch an approach to use the recovered AD to extend the system. Then, the use case will be wrapped up.

6.2.8 Extending the PowerGlove

The created AD could serve as a starting point for extending the system. For example, in adding *grasp support* to the system.

The extended PowerGlove adds to the patient's concerns that the device must support the grasp of the user by supplying a force that amplifies the subject's neuromuscular intent to grasping.

This section is for illustrating the starting point of a possible approach to extend a system using its AD. To actually do this, a more throughout analysis of stakeholders and concerns should be conducted. The focus of this section is on the process, not the result. The production cell use case provides a more extensive demonstration of extension.

A suitable scenario, based on the extra concern of the extended PowerGlove, consists of a user, wearing the glove, that tries to grasp a glass:

S2.4 - Grasp Support

- The user wears the device in a clinical setting during performance assessment.
- The user grasps a glass wine, beer glass that is empty, full.
- ✓ The extended PowerGlove should apply a supportive force to the grasp.
- ⚠ The scenario fails is the glove does apply supportive force when NOT grasping.

This is a still a crude scenario but it suits this exploration phase and is still extendable for later development iterations.

Figure 6.10 shows a possible extension of the logical view of the extended PowerGlove. Additions for the extended PowerGlove are yellow-coloured. From the scenarios and the new concern, the main goal is derived. The corresponding task is *strengthening grasp*, which reuses the *estimate hand pose* task. The task relies on *grasp detection* and *grasp force production* skill which in turn require some basic services and functions as the diagram clarifies. The logical view helps in reusing functionality of the existing PowerGlove. A downside is that it is not clear from the view whether the functionality is located at the same physical location, this follows from other views.

Through careful evaluation of the correspondences between the logical view and the other views and the addition of missing elements to these views, the system can be iteratively further developed.

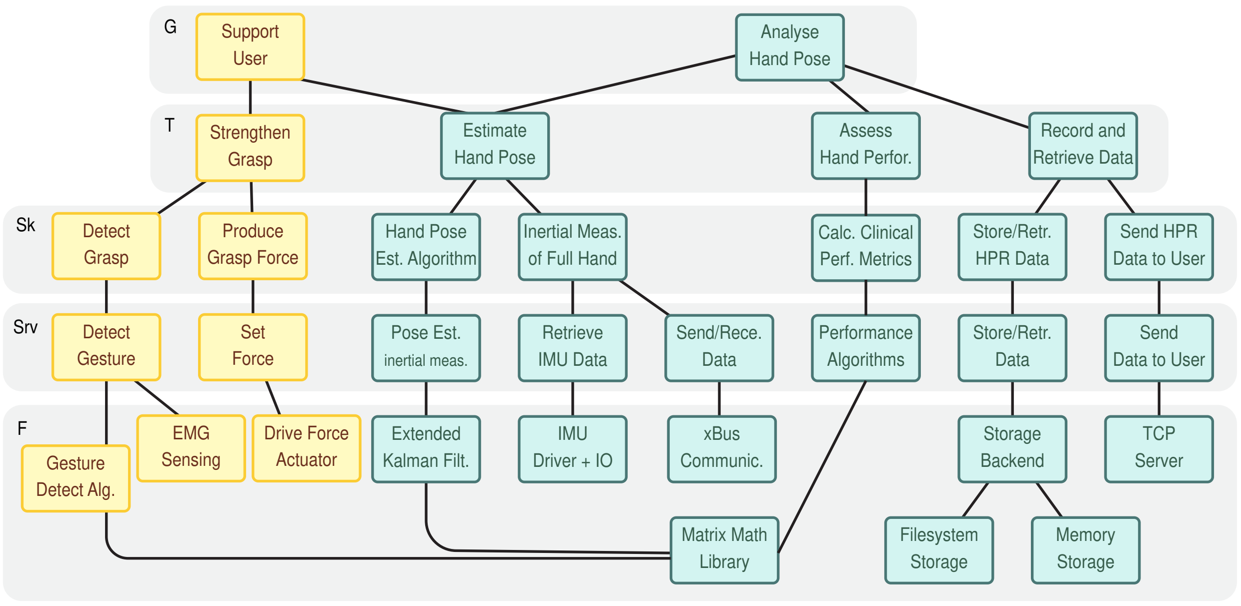


Figure 6.14 Example of the logical view of an extended PowerGlove. Additions are marked yellow.

Wrapping up

The advantages of the architectural approach in this use case are:

- Ability to express an existing architecture to improve its understanding and documentation
- The logical, process, deployment and development viewpoints allow expressing the whole system from relatively independent perspectives.
- Scenarios force the user of the framework to explicitly think about the purpose of the system.
- New functionality can reuse existing functionality.
- Small development iterations reduce development risks and ensure consistency throughout the development.

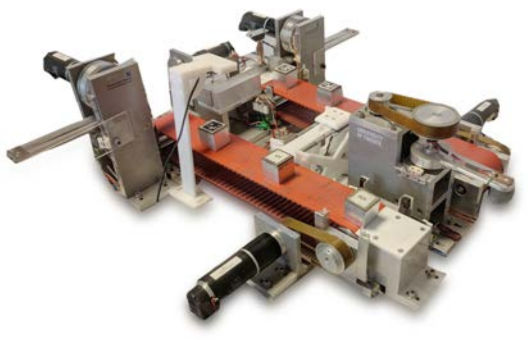
In this use case, the main disadvantages are:

- Difficult to simulate and test the system in the early phase as complex models of the environment are required.
- Process view relies on abstractions that do not always hold, requires patching that may lead to confusion.
- Scenarios provide consensus but poorly constructed scenarios provide confusion. Designing good scenarios can be difficult.
- Not all possible scenarios can be exhaustively tested, so the selection of the appropriate scenarios remains a design problem.

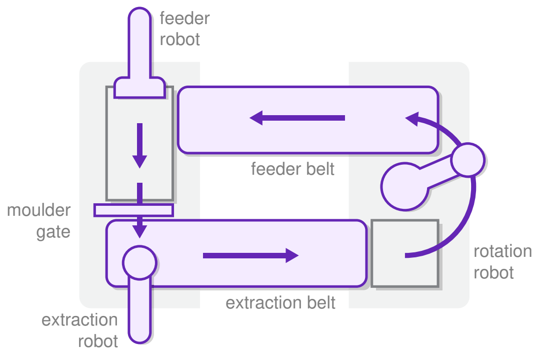
In terms of **understandability**, the framework provides documentation, scenarios and detailed information about the system's implementation.

The **extensibility** of the system benefits from the logical view, which provides an overview of all functionality available in the system and their links to corresponding elements in the process an deployment view. In case of the *grasp support* extension, the functionality was added to the logical view through the addition of a concern and a corresponding scenario.

The **reusability** is parts of the system improves from the presence of an AD as it allows to analyse an structure the functionality and features already available.



(a) Bird's eye view of the *Production Cell Setup*



(b) Abstract view of the production cell's components with the processing direction indicated.

Figure 6.15 The Production Cell

6.3 Production Cell Use Case

The *Production Cell Setup* [70] is a scale model of an industrial plant. It serves as an experimentation platform for mechatronics and CPSs featuring multi-domain sensors, actuators, a mechanical setup, power electronics and an embedded control system. Figure 6.15a shows a bird's eye view of the system and figure 6.15b shows a simplified top-view of its mechanisms.

The platform is the result of researchers and students' efforts at the University of Twente and intended to facilitate CPS research. In their 2009 paper, Groothuis and Broenink explored the software and hardware design space of the setup. The result consisted of seven motion control system implementations which build upon or relate to the CSP language. In 2015, Vos extended the production cell with a sorting module, Robot Operating System (ROS) and TERRA-LUNA support. Then, in 2018, Ridder improved the setup and corresponding software. Results of Ridder's improvements, include a simulation in ROS' Gazebo simulation environment; TERRA-LUNA support for Functional Mock-up Units (FMUs), ZeroMQ messaging and ROS communication support.

The production-cell platform has the typical multi-domain and multi-node characteristics of a CPS: Distributed control, communication and orchestration contribute to the *technical emphasis* of the platform. The *level of automation* of the platform reflects in its ability to manage and orchestrate itself, mostly without human intervention. The system is subjected to continuous development and improvements and this benefits from a *life cycle integrating* development strategy that produces artifacts such as simulations, tests, documentation and publications. The coupling between control performance, mechanical design, orchestration and communication demonstrate the *cross-cutting aspects* of the platform.

This use case builds upon the results of these previous researches to show how to incorporate existing efforts to speed up development. Among these previous results are a simulation model and a physical prototype. Because of these previous efforts, this iteration is enumerated *N*.

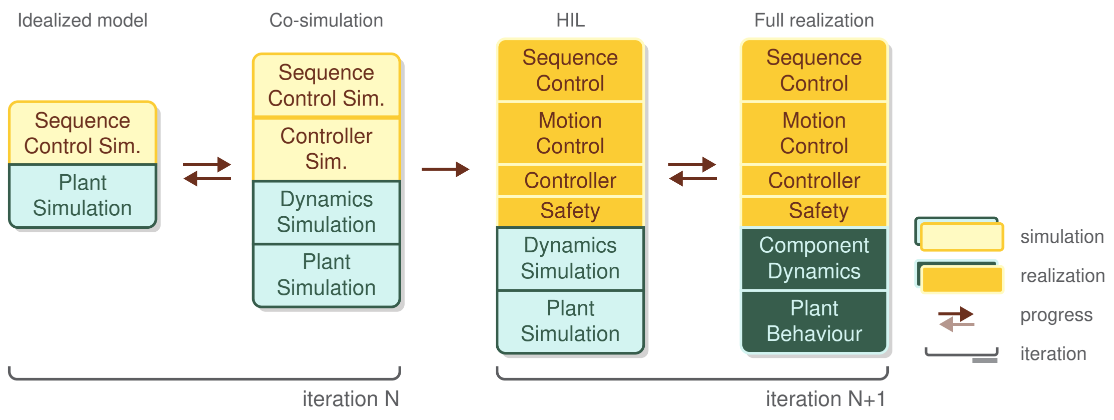


Figure 6.16 Development process of the *Production Cell Setup* in the AF. The software simulation is executed in a Linux soft real-time LUNA environment whereas the realization is a hard real-time system. The hardware simulation is executed in the Gazebo simulation environment.

In this - more extensive - use case, all domain concerns are involved. The fact that the architecture will be tested on a real system, makes this use case more representative for an actual CPS than the first use cases. In the process of distilling goals and requirements, the **effectiveness** of the framework's contribution can be evaluated. The framework's contribution to the **modularity**, **reusability** and **extensibility** is discussed. **Testing** and **consistency** will be topics of this use case as well. Finally, the way in which the framework helps to address the **simplicity** and **understandability** concerns are discussed.

A summary of the iterations is, in advance, shown in Figure 6.16. The first iteration (N) aims at setting out and simulating the preliminary system architecture and implementation. First, using an idealized system that does not involve any component dynamics but rather *enforces* the setpoints on the plant. Then, the architecture is extended by including component dynamics and controller to the simulation. A fully working (co-)simulation is the goal of the first iteration. This co-simulation might help in identifying issues and possible solutions.

Throughout the life cycle of the system, development iterations will result in improved versions of the architecture that succeed previous ones. Newer iterations will typically *improve* functionality and performance and reduce issues. However, where possible, track these earlier versions together with their corresponding architecture decisions and rationale. Older versions of architecture can still provide useful insights and information about the system.

The following sections describe the AD of the production cell (iteration N): stakeholders and concerns; scenarios; the logical, process, deployment and development views.

Architecture decisions are tracked in the digital software/model repository in which the AD will be tracked too. A copy of the ADRs is provided in Appendix D.3 and to specific ADRs is referred to using the `ADR-PC-0000` notation where the number corresponds to a single ADR. The first decision involved in the design of this use case is the choice to track ADRs using the Markdown Architectural Decision Record (MADR) convention (see `ADR-PC-0000`).

6.3.1 Stakeholders and Concerns

The purpose of the system is to ‘mould’ blocks in a way that satisfies its stakeholders. A stakeholder-concern analysis will help in determining what the stakeholders would satisfy.

The first step is to determine the involves groups of stakeholders. The AF suggests three classes of stakeholders (chapter 3) that are the starting-point of the analysis.

Architects, Designers and Developers are those that involve in the analysis, control and documentation of the system.

Builders and Maintainers care about the feasibility of the system and the maintainability of its components.

End-users care about the performance and cost of the system and the effort involved in operating it.

The general concerns of the domain stakeholders are listed in the AF. Additionally, the following specific concerns are identified:

Self-dependence The system must operate with as little supervision as possible (end-users).

Productive The system must be productive in moulding blocks (end-users)

Compatible The system must be compatible with its intended environment (builders, end-users)

Note that the use case involves a scale model and that full system is much bigger. Compatibility with the intended environment thus boils down to compatibility with (the scale of) a factory floor. These system specific stakeholders and concerns form together with the standard the full set of the production cell in this iteration.

> It may be difficult to distil all relevant stakeholders and concerns from the information available to a user of the framework. A somewhat crude initial guess will provide a basis for the first iteration, the outcome of which can be used to review the selection of stakeholders and concerns in later iterations.

6.3.2 Scenarios

Scenarios describe a situation and the expected behaviour of the system.

The *normal production operation* is a very important scenario that describes the situation in which the system functions as intended. Situations in which the system does not function as intended must be evaluated too, to analyse the consequences of these situations and how to handle them. The most probable scenario that describes a situation that is abnormal, is that a component of the system halts, causing any blocks in that component to halt. In this scenario, the system must ensure that *feeding* components do not overload the halting component and cause collisions.

These basic scenarios will allow an initial problem analysis and a crude solution design. The ability to simulate scenarios can be helpful during analysis, design and test. To enable this, a simulation environment must be set up that is suitable for the kind of system under consideration: a multi-body three-dimensional mechatronic CPS.

Simulation Environment

To make such a scenario testable and preferably automated, a suitable *test* environment is required. A complex physical system like the production cell undergoes more interactions than can conveniently be mathematically described. Especially the physical interactions of non-rigid (realistic) multi-body are difficult to model analytically. An outcome to work with such systems is the use of a multi-body physical simulation environment and Gazebo is such an environment. Gazebo is part of the ROS ecosystem and as such is highly integrated into the robot development system. The use of Gazebo in this iteration is decided in `ADR-PC-0001`.

A suitable simulation of a complex system requires modelling of its physical components. Though this may sound like a difficult extra effort that is not necessarily true: especially the components of the system that need manufacturing will be modelled digitally in the first place. Creating simulation models is then a matter of translating these digital models which is rather trivial with modern simulation software that supports importing and exporting a wide range of models. In the case of the production cell, the components are modelled in the SolidWorks CAD application and then imported into the Gazebo simulation system.

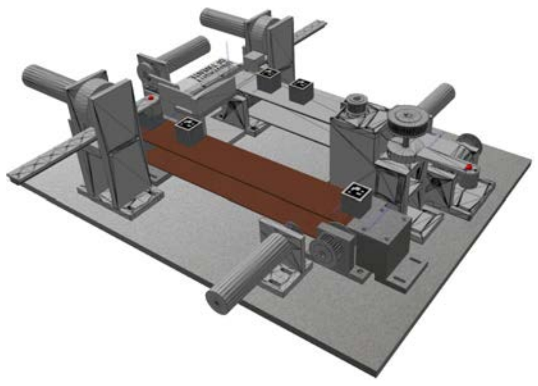
The resulting model is shown in Figure 6.17a and the striking similarity to the actual production cell (Figure 6.15a) is promising. Figure 6.17b shows a sample of the simulation interface available to the user.

An attractive aspect of the simulation environment is that it can be interfaced with using the same control software as can be used with the actual plant. Another advantage of the simulation environment is that measurement of states and variables is not limited to those that have a corresponding sensor attached. This allows better analysis of the simulation.

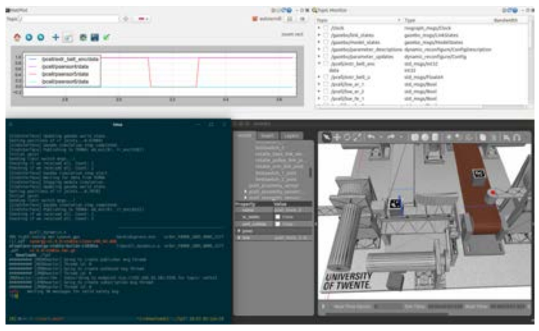
The following section discusses the scenarios and their corresponding Gazebo model.

Normal Production Operation

The *normal production operation* scenario describes a situation in which the production cell operates to produce *blocks*.



(a) 3D overview of the model



(b) Sample of simulation interface. LRUD: plot traces; list of topics; command console; Gazebo visualization

Figure 6.17 Simulation model of the *Production Cell Setup* in Gazebo

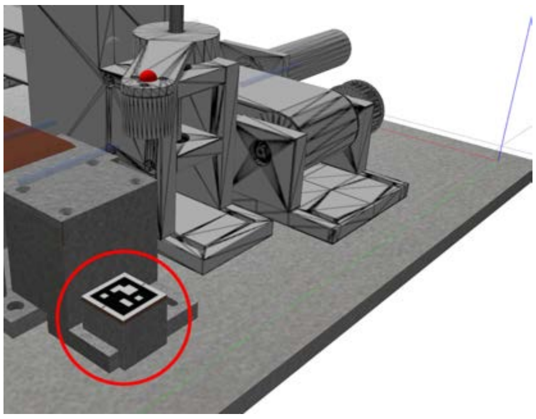
S3.1 - Normal Production Operation

- This scenario describes the full production cell in a standby situation
- Zero to eight objects are placed in the production process, spaced at least half a length of an edge
- The production cell is given the START signal
- ✓ The system should process all items
  - Blocks are fed to the moulder
  - Blocks are moulded
  - Blocks are removed from the moulder
  - Blocks are stored
- ⚠ The system fails if any operation on any block fails, i.e., if a block gets lost in the process or stuck in a component.

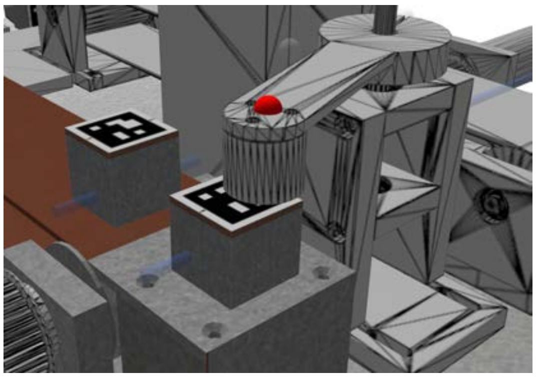
To enable the initial situation, the Gazebo simulation environment is set up. The output of the simulation is exposed on the ROS communication bus which allows a simple listening ROS node to determine whether the scenario succeeded or failed. The Gazebo simulation timescales with processing power and can potentially run faster than real-time, enabling test speed-ups. Figure 6.18a visualizes scenario S3.1 and shows a situation in which the scenario test fails.

Collision Avoidance

An important aspect of the system is that collisions should be avoided. This test describes the situation of collision and how the system should (not) act to resolve such a situation. Figure 6.18b visualizes this scenario.



(a) A scenario that describes that blocks must not leave the belt. In the situation pictured, the scenario failed.



(b) Visualization of a scenario in which a collision is tested.

Figure 6.18 Visualization of two scenarios of the Production Cell

S3.2 - Collision Avoidance

- This scenario describes the logistics unit and the moulding unit.
- At any pick-up or drop-off position, a block is put and hold.
- Another block is in the system too.
- ✓ The system should avoid collisions by waiting until a position is free
- ⚠ The system fails if an object gets stuck or leaves the process

6.3.3 Logical View

Using the concerns and the proposed scenarios, the logical view can now be created. By evaluating the stakeholders, concerns and scenarios, a first goal is extracted: *Produce Moulded Blocks*. This rather straightforward core goal can satisfies the concerns and fits the proposed scenarios of the production cell and is, therefore, a suitable starting point.

This goal can be decomposed into tasks and consequently skills. Two tasks support the main goal. First, *Block Moulding Operation Control* to ensure that each component operates as intended. The scenario describes that collisions should be avoided while the concern of *self-dependence* was posed. A design question that comes to mind is: what type of coordination between components is most suitable. This answer is decided in `ADR-PC-0004`: decentralized and opportunistic to allow subsystems to function independently of other systems. Only some supervision and interaction is required to orchestrate the system. That means that the subsystems should coordinate.

The *Supervision and Interaction* task takes care of this coordination of the system as a whole. Interaction was not explicitly listed as a concern but an architect might suspect that the end-user still wants to

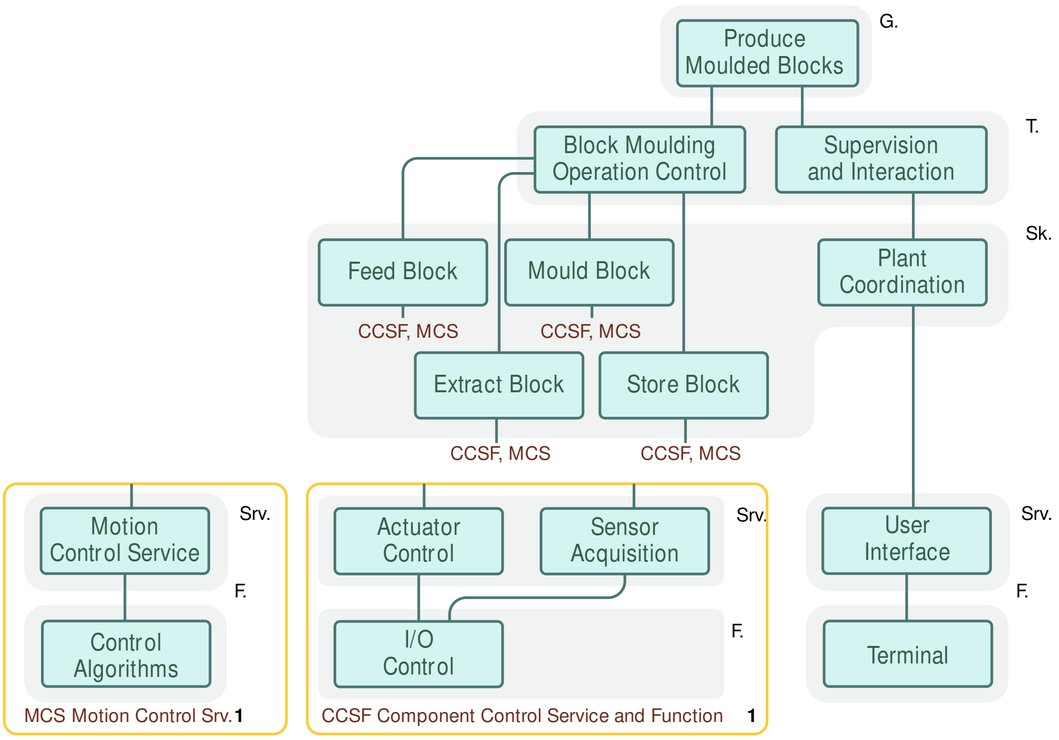


Figure 6.19 Logical view of the Production Cell

start, pause and (emergency)stop the system, so this is decided to be included in the supervision task.

The user of the framework is the architect of the system and uses the input from stakeholders to coordinate the creation of the architecture. The architect must realize that this input is a guideline and that their freedom remains to add or alter the system as they see fit. If it were not for these difficult situations that require insight and analysis, there would not be much for an architect to do in the first place.

The hierarchy is further decomposed. The *Block Moulding Operation Control* task depends on operation skills that enable the task: moulding a block, feeding the moulder, extracting from the moulder and logistics management.

The *Supervision and Interaction* task is enabled by a plant coordination skill and communication with the *Block Moulding Operation Control* task.

Finally, the skills are supported by services and functions. For now, the skill-elements will be supported by a set of services and functions that help in plant control whereas the plant coordination and plant supervision skills will be supported by networking services and functions. In this way, the plant control functionality is decoupled from the application and can be reused or developped independently. Appendix F provides information about the control library used. Figure 6.19 visualizes the logical view described above.

In an early iteration a concise description of the functional elements is often sufficient because of two reasons:
First, it is hard if not impossible to determine all details of the system in such an early stage of development; Second, defining all functional elements that support the full implementation of the final system is often not necessary in prototypes and simulations.

6.3.4 Process View

The process view comprises the behaviour of the system through compositions of processes in different MoCs.

The previously discussed AEs guide the creation of the process view. As indicated by the AF, this typically starts by determining how the topmost goals must coordinate the corresponding tasks to be successful and then repeat this for the lower levels.

The *logical view* gives an overview of what the system needs to achieve. To enable the *Produce Moulded Blocks* goal, both supporting task must be active throughout the operation of the system. The supporting skills need to be active throughout the operation too. This can be achieved using parallel processes or repetitive sequential processes. For this global coordination, a Communicating Sequential Processes (CSP) MoC is selected that - based on experience - integrates well with this type of system and available tools (see `ADR-PC-0003`).

Within this base MoC, the processes that enable the tasks and skills must find a place. The parallel nature of the production cell's components favour parallel CSP processes for sequence control and decentralized opportunistic control. The information required from sensors and the control setpoints are modelled as CSP readers and writers. The sequence controllers have a state and event nature so they might be modelled as FSMs. The output of the sequence control section needs to lead to actual control of the system. An actuator control CSP process is added with a similar structure but with discrete time MoC for the controllers instead.

Figure 6.20 visualizes the structure of the process view of this iteration.

The discussed process view will be implemented in the TERRA-LUNA environment because of its support for the relevant MoCs and integration with available hardware (see `ADR-0002`). These models are configurable, editable and automatically generate corresponding program code.

The process view does not model the behaviour of the mechanical system as the mechanical design freedom in this use case is limited and appropriate information about the mechanical behaviour can be retrieved from the scenario simulations.

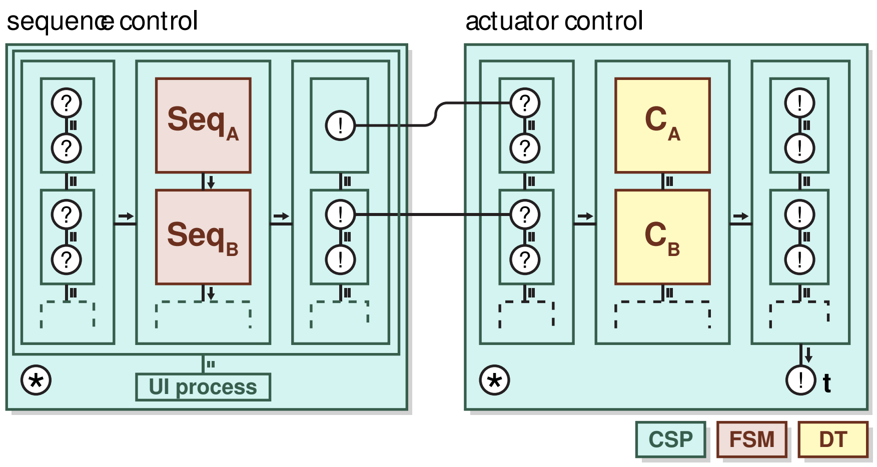


Figure 6.20 Overview of the first iteration of the process view of the production cell setup. The process view denotes a composition of CSP, FSM and DT MoCs.

6.3.5 Deployment View

The deployment view describes the structure of the system under consideration. This view involves the relation between physical components of a system and the goals and tasks that they need to achieve cooperatively.

The goal is to determine the connection and structure of the physical and cyber parts of the system to enable the goals and tasks of the system.

The *compatibility* concern might influence this view significantly: the use case represents a scale model which should be designed for compatibility with the scale of a factory floor. That means that using a single cyber-node at the centre of the system may not be practical. So, two cyber-nodes will be used. One at each end of the production cell, connected by a communication medium. Furthermore, all sensors and actuators connect to the corresponding cyber-node. This provides us with an initial deployment view for the production cell. Figure 6.21 visualizes the deployment view of the production cell in this iteration.

The deployment view is also implemented in the TERRA-LUNA environment though this environment has some limitations with respect to the components that can be modelled. Physical components, external nodes and communication links between nodes are not well described by the TERRA-LUNA framework.

6.3.6 Development View

The last view of the architecture to be constructed in this iteration is the development view that models how the cyber-nodes should be implemented to support the structure and behaviour described in the other views.

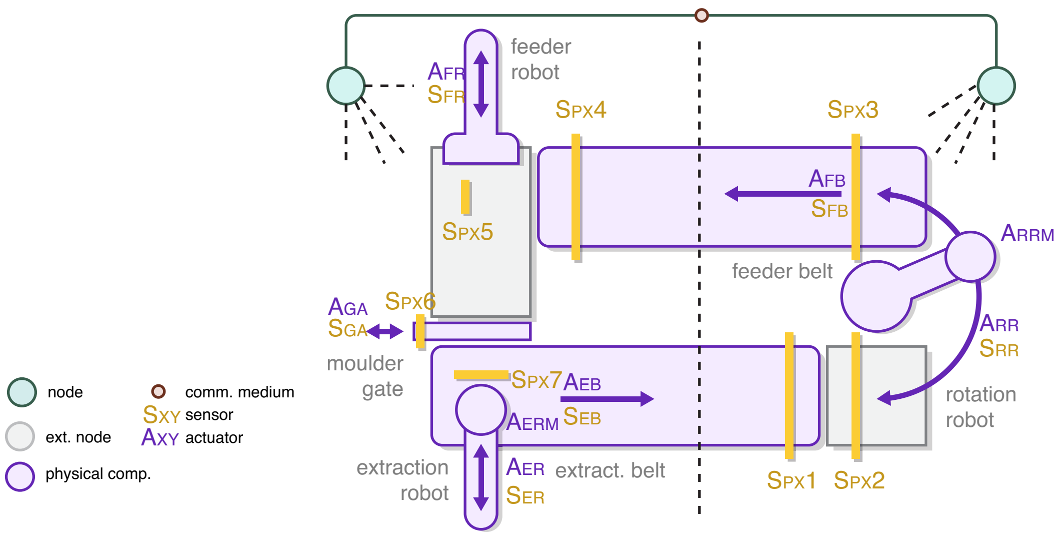


Figure 6.21 Deployment view of the production cell, all sensors (prefixed S) and actuators (prefixed A) are connected to the cyber-node at their respective section. `ga` gate; `er` extraction robot; `erm` extraction robot magnet; px proximity sensor; `fr` feeder robot; eb extraction belt; `rr` rotation robot; `rrm` rotation robot magnet; `fb` feeder belt.

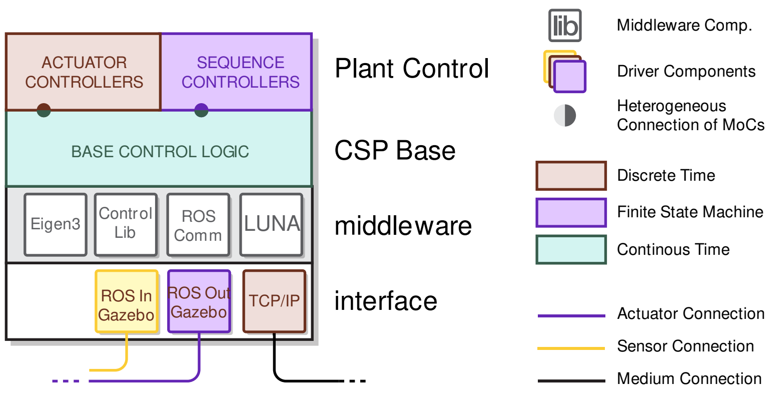


Figure 6.22 Development view of the production cell, using ROS communication

Also, decisions about what libraries and implementations to use belong to this model.

Figure 6.22 visualizes the development view of the production cell. The result is an overview of the structure of the cyber-nodes to be implemented.

The production cell uses a real-time execution framework *LUNA*, a library with control algorithms and a ROS communication library.

6.3.7 Putting it all together

The models of the process view and the deployment view where modelled in the TERRA-LUNA environment. To create actual executable applications, the development model needs to be modelled in the

TERRA-LUNA environment too. Currently, there is no support for modelling such a development model, so this had to be done by hand:

- Configure the MoCs and their coordination
- Configure drivers for communication, sensors and actuators
- Link all libraries appropriately

In the case of the production cell, this is done through a combination of hand-coding, code-generation and build automation. This process resulted in executable binaries that represent the corresponding cyber-nodes.

The sensor and actuator interface was chosen such that these binaries can be combined with a simulation using ROS Gazebo. A simulation was set-up and configured - using technologies developed by Ridder [72] - to assess the result of the architecture. This simulation setup allows to develop and test preliminary developments before applying them to a physical setup. In the case of high-cost or sensitive systems, this can be a useful alternative to testing on the physical system.

Though the simulation environment was successful in this use case, the technologies used are still far from perfect. To name some significant issues:

- The performance of the TERRA-LUNA / ROS Gazebo setup was very low, about 0.1x normal time.
- Because a standard interface between applications and technologies lacks, creating co-simulations often requires custom created simulation interfaces which are tedious to create and error-prone.
- Representative models of the scenario need to be created which is often unfeasible.
- A simulation is not the actual system and it is unclear what discrepancies between the simulation an reality mean to the results.

The alternative in these situations is to test and validate parts of the system digitally and execute an integration test on a physical prototype or system.

The results of the integration of the architecture helped in testing the sequence control processes. Another result of the simulation is that the control effort is too high when using step-shaped references. Also, the actual production cell has no absolute position sensor - as opposed to the simulation - but only limit-switches. Homing functionality is required to make the production-cell work. Furthermore, the evaluation of the architecture led to a new concern: safety.

The following section starts a new iteration to improve the production cell architecture.

6.3.8 Continuing development, Iteration N+1

Iteration N+1 takes the development of the production cell further by improving the motion controllers, introducing safety functionality and realizing the architecture on the physical setup. To ease the transition from simulation to realization, the setup goes through an intermediate step in which the cyber-nodes are realized on hardware but the plant is still a simulation. The main benefit of this Hardware in the Loop (HIL) approach is the ability to find controller realization issues and bugs without risking the actual system which increases system safety and decreases realization costs.

6.3.9 Stakeholders and Concerns

The previous iteration raised a safety concern, and to take care of it, it is added to the system's AD.

Safety All stakeholders care about the physical safety of the users that work with the system, may it be because of direct danger or indirect responsibility. The system should be able to handle emergencies.

6.3.10 Scenario Viewpoint

A scenario is added to the architecture that represents a situation in the safety concern is very relevant:

S3.3 - Emergency

- The system is in any state, stopped, paused or running.
- Any controlling cyber-node detects an emergency
- The cyber-node triggers an emergency signal
- ✓ The system should halt within 100 ms
- ⚠ The system fails if it does not come to a halt within 100 ms
- ✓ When the system is not sure whether there was an emergency (no communication possible) it should also halt.

6.3.11 Logical View

The logical view is now adjusted to accommodate new and modified functions. Figure 6.23 visualizes the modified logical view.

- Safety features are added. The *supervision and interaction* tasks could be made responsible for the emergency feature. The plant is extended with an additional supervision skill and a communication service. A requirement is added to the *plant supervision* skill: maximum 100 ms between an emergency signal and actual plant stop.

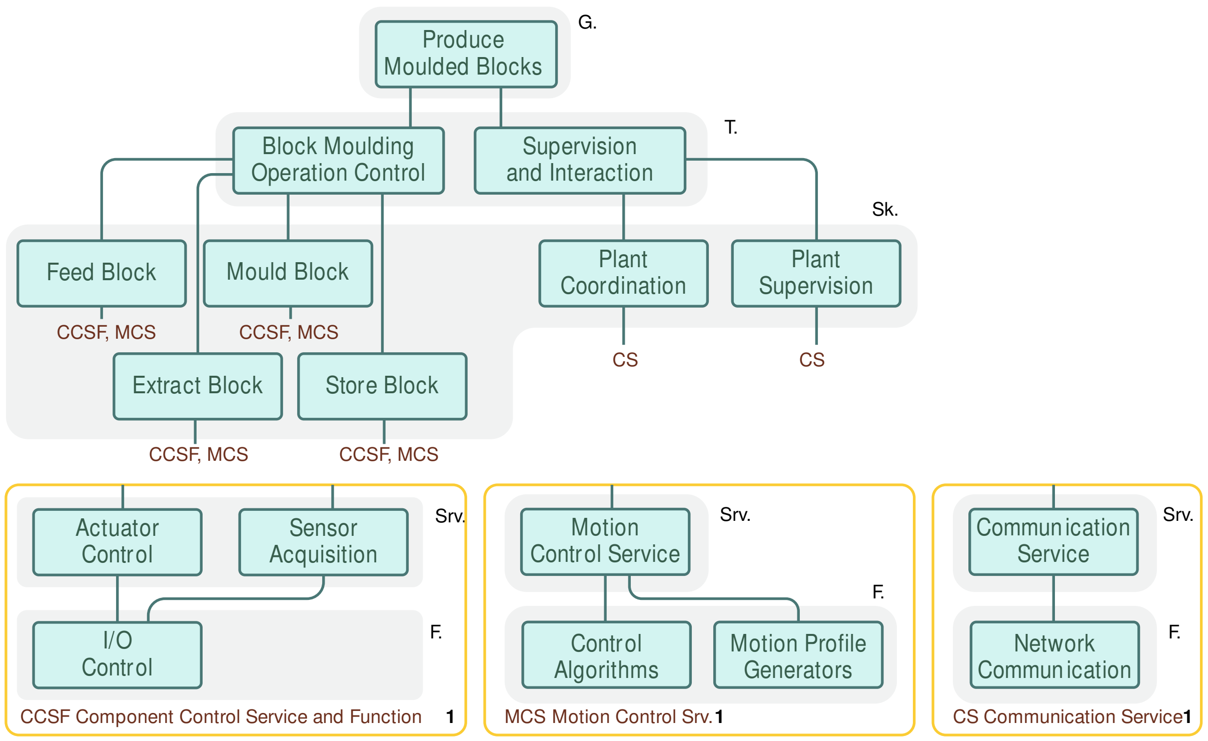


Figure 6.23 Extension of the logical view in the second iteration. Includes communication service, plant supervision and motion profile generation.

- The control-effort of the first iteration was too high. By using motion-profiles, the peaks of step-shaped setpoints can be avoided. The *motion control service* is extended by adding a *motion profile generator* function.
- A homing procedure is required to obtain the absolute position of the actuators. This functionality is in the scope of the *feed*, *mould*, *extract* and *store* skills.

6.3.12 Process View

The modifications of the logical view must now be implemented in the process view. To account for motion profile functionality and homing operation, a new process is added as an intermediate between the sequence and actuator control processes. Figure 6.24 shows this extension. Both homing sequences and motion profile generators are added to a CSP process. To allow switching between the *homing* and *motion profile* state, an *alternative* construct is used (indicated by block square in the centre).

The safety functionality is implemented in the lowest level process: the *actuator control* process. To ensure that the system passes the scenario and requirements, the CSP process was modified with two sequential safety processes. The safety processes send a status message with a sequence number to the *next* node in a chain. The *next* node checks the sequence number and the state of the message. If both sequence number and state are fine, the processing is continued. If the status

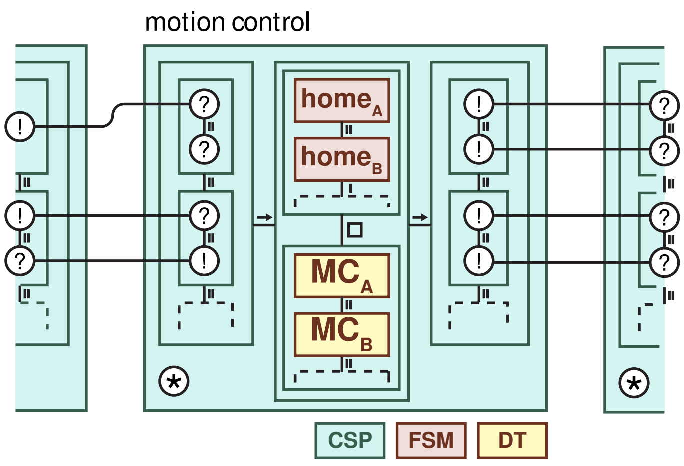


Figure 6.24 The process view is extended with a motion control model that takes care of homing and motion profile generation.

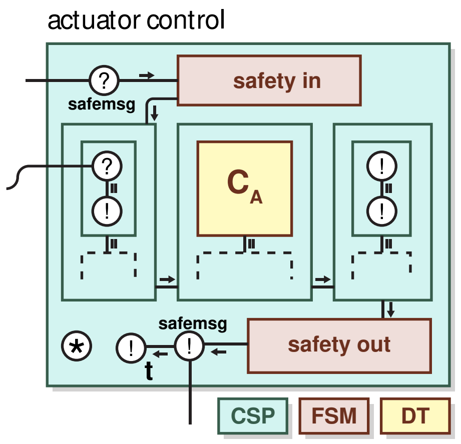


Figure 6.25 To ensure safety, a mechanism of exchanging safety messages is implemented in the low-level controller. The controller does not block on messages.

is *emergency*, the node stops operating and forwards the message. If no message arrives with an expected sequence number within 100 ms, the node also shuts down. The system recovers when all nodes are fine and the sequence numbers of messages are correct. Figure 6.25 shows the modified *actuator control* process.

6.3.13 Deployment View

The deployment view does not need significant modifications in this iteration, except for a communication line between cyber-nodes that allows the exchange of status and safety information and the addition of two limit switches at each actuator to enable determine the absolute position through homing.

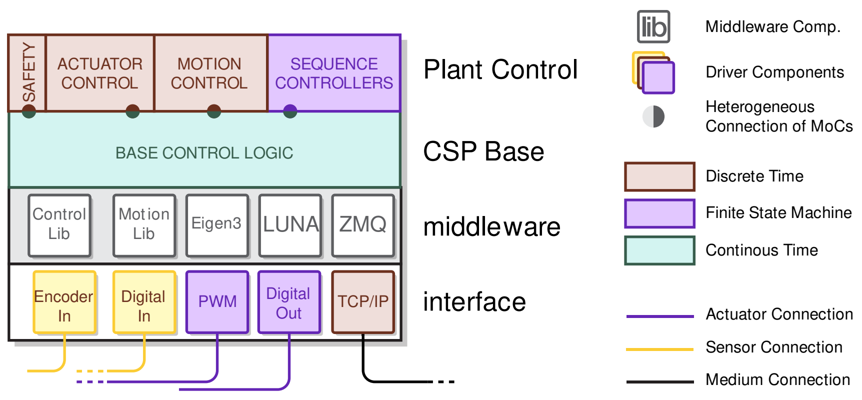


Figure 6.26 The updated development view of the Production Cell. The ROS communication is replaced by actual sensors and actuators

6.3.14 Development View

The development view of the system is now adjusted to incorporate the changes in the process view: addition of a motion control process, a safety process, a motion profile generation library, ZeroMQ libraries. The system can be tested with the previously discussed simulation environment and if that suffices, it is time to test it on the physical plant. To do this, only the connections in the development view model and corresponding parts of the system need modification. The resulting development view is shown in Figure 6.26.

6.3.15 Concluding the iteration

Once again, the updated models can be integrated to test the architecture. First, in simulation, and if that succeeds, on the physical system.

Moving from simulation to real system involved some difficulties: the mechanics of the simulation behave not exactly as the actual mechanics so the tuning of the controller and motion parameters was off. This, however, was expected and resolved through the well known straightforward trial and error process of parameter tuning.

The resulting system can be witnessed on video, see the link below.
`https://youtu.be/8rRGD0P6vJM`

Wrapping up

This use case discussed two iterations of applying the AF to a design problem.

Code generation played an important role in the realization of the system from the AD. Though code generation makes the realization process initially faster, it will require extensive knowledge about the underlying implementation as soon as something works differently than expected.

Because the mechanical design of the production cell already existed, its design was not part of the design process. The framework provides only few tools and means that help in the mechanical design of the system.

The main advantages of the framework in this use case are:

- Architecture helps in reasoning about possible structure of the system.
- The framework supports an iterative design process, combined with HIL techniques, to streamline the transition from simulation to realization.
- The production cell could be simulated as a whole to enable a *safe*, reusable testing environment.
- The *meat* of the architecture is tracked in the form of Architecture Decisions (ADcs) and linked to the corresponding Architecture Elements (AEs).

The subsequent disadvantages of the framework in this use case are:

- No tools available to model and realize the development view.
- Integrating all models to enable simulation requires writing custom wrappers and coordinators which is tedious and error-prone.
- Though the mechanical system design was already determined, the framework provides very few means to design and analyse it.

The structured analysis of stakeholders and concerns helped in determining scenarios and correspondingly the required functionality and components for an **effective** system. Integrating and simulating the system at the end of an iteration helps in assessing its performance and indicating whether all requirements are met. In the use case, the integration of the first iteration led to a new concern, that of *safety*, and the framework allowed to translate this concern to a corresponding effective architecture.

By separating application specific goals and tasks from generic functionality and components, the **modularity** and **reusability** benefit. For example, the control libraries and I/O interface were decoupled through logical *services* from the *goals, tasks* and *skills* of the system.

The help that the framework provides with **extending** is demonstrated in iteration N+1, in which *safety signal* functionality is added.

A **testable** system typically results in fewer problems during run-time. The framework provided means to improve the testability of the CPS. The downside of these means is that they require suitable models of the system and its environment which are not always available or feasible.

The framework supports the **consistency** of the production cell architecture by providing rules to check for consistency between architectural elements and an iterative design process with integration both early and often through scenarios. In this use case, testing and consistency checking helped in detecting implementation issues like unsafe control efforts and the absence of absolute position sensors next to the big number of smaller and general bug-fixes.

The **understandability** of the system benefits from the availability of an AD that consists largely of models that are used in the actual system too. Also, the tracking of architecture decisions and rationale helps in improving the understandability of the system.

Whether the **simplicity** of the system benefits from the framework is hard to subjectively determine but rather a matter of preference. The framework tries to steer towards a goal-oriented and iterative development process while keeping the AD concise and to the point. Some designers, however, might prefer a purely computer-model based approach that does not involve a stakeholder-concern analysis and the specification of scenarios.

A video recording of a demonstration is available online.
`https://youtu.be/8rRGD0P6vJM`

7 Discussion

The design and management of CPSs is complex and a bottleneck in their adoption. Though industry and academics are putting much effort in CPSs research, a need for stronger modelling semantics remains. This research aimed at developing an Architecture Framework (AF) to improve the design and management of CPSs in research projects in order to create better CPSs. The goal was set out in four research questions that yield the elements of the AF:

- Q1 What is a typical CPS: rationale, chapter 2
- Q2 What aspects *improve* the design and management of CPSs: stakeholders and concerns, chapter 3
- Q3 What are suitable viewpoints to cover these aspects: viewpoints, chapter 4
- Q4 Does the resulting AF indeed *improve* CPSs: chapter 6

The following sections discuss these research questions.

7.1 What is a Typical CPS

In chapter 2 the main characteristics of CPS were extracted to answer the research question ‘What is a typical CPS’. A clear separation between *cyber* and *physical*; with a hierarchical structure of information flow between these worlds was identified. This notion of separate worlds builds upon the assumption that they can be modelled relatively independently but in the end, the cyber world consists of physical elements. So, strictly speaking, there is no real independence but only approximated independence. To make sense of the physical world, the process of information extraction in a CPS was identified and split into three phases: *experiment*, *theory* and *computation*. MoCs are the generalization of modelling conventions in which these phases work and interconnecting different kinds of MoCs is the foundation of heterogeneity. The physical components of a CPS were determined: *cyber-nodes*, *sensors and actuators*, *communication media* and *external nodes*. All abstractions, however, suffer to some degree from *leakyness*. This comes forward in the abstraction that the cyber world and the physical world solely interconnect through sensors and actuators. Situations in which the physical world does affect the cyber world and vice versa are not modelled intuitively in this framework. To account for this, physical components or communication media dynamics must respectively represent the influence of cyber-components on the physical world and the influence of the physical world on the cyber world. This limitation is the price we pay for the benefit of being able to decouple the two worlds. The result of this analysis is a description of a *base* CPS that essentially is an abstraction of specific CPSs and that provides an interface to these specific CPSs without having to concern about their implementation.

We did not decide on a hierarchy of information in the cyber-world. Such a hierarchy, however, seems to exist in most systems. The ability to reason about the level, quality, and hierarchy of information could be a useful concept to improve the power of the architecture framework and corresponding architecture descriptions. A possible solution to this may be found in the field of *ontologies*. This is a topic for future research.

Composing different models of computation is essential in the framework. This is a non-trivial problem which involves weighing different methods by their advantages and disadvantages. We discussed two solutions for the simulation of heterogeneous models in the appendix that have different benefits and problems. A de-facto solution to heterogeneous composition does not yet exist.

We proposed the use of graphs as a possible mathematical notation of relations. These graphs are powerful in that they are combinable and analysable with graph theory. Such an analysis might automate parts of the design procedure by, for example, optimizing the deployment or restructuring the development model for cyber-nodes. Thorough formal validation of architecture descriptions belongs to the possibilities. We did not yet explore this possibility and propose this to be a topic of future research.

7.2 What aspects improve CPS’ design and management

Good practice in systems and software engineering is to first determine which parties are involved in a system and what their concerns are. This AF sees to this through the adoption of stakeholders and concerns as primary design artifacts. Chapter 3 translated *aspects that improve CPS’ design and management* into eight concerns to answer question Q2 in a form that is compatible with an AF. This choice of domain concerns is based on literature review but is extensible by the user of the framework to allow the addition of system-specific concerns. It remains questionable whether the proposed selection of concerns is *best*.

7.3 What viewpoints cover these aspects

Chapter 4 explained the viewpoints that make up the core of the AF. These viewpoints answer research question Q3 and are formulated to comply to the IEEE-42010 standard.

The scenario viewpoint aims at describing the intended use of the system and the interaction with its environment. The logical viewpoint provides a perspective on the system aimed at distilling the functionality and required components from concerns and scenarios. The process viewpoint focusses on the description of behaviour of the physical and cyber aspects of the system. The deployment viewpoint targets the structure and connection of the physical components of the system. The development viewpoint provides a perspective on the structure of the cyber-nodes of the system.

In the use cases, these viewpoints showed useful to describe different systems. The viewpoints helped especially in the creation and design of cyber-nodes that interact with the physical part of the system and the environment. The design of the physical aspects of the system themselves, like mechanics, is more difficult using these viewpoints and relatively underexposed in the use cases.

The separation between behaviour - through the process viewpoint - and structure - through the deployment viewpoint - can be troublesome as structure and behaviour are physically tightly related. This results in coupling between the two views that might result in inconsistencies when integrated. Currently, early and often integrating the views is the proposed solution to this possible issue.

7.4 Does the AF improve CPS' design and management

In this research, we discussed the application of the architectural approach to three use cases. These use cases serve two purposes: they help to show whether the AF indeed *improves* CPSs (question Q4) and they serve as example applications for the AF. Especially the third use case is useful because it applies the AF to a real-life system that represents a typical CPS and will, therefore, do a better job at pointing out issues with applying the framework. These use cases, however, provide only anecdotal 'proof' of whether the framework is useful in practice.

7.4.1 Reviewing the Concerns

Section 3.2 listed the domain concerns of the AF and this section will discuss how the AF addresses each of these concerns. The starting point per concern is an explanation of what extra handles, tools and methods specific to this concern the AF provides. This then leads to an explanation of the advantages and disadvantages of using the AF for addressing the concern.

Effectiveness

The framework supports the effectiveness of CPSs by providing a structured way to work with high-level concerns and goals through the stakeholders and their concerns and the decomposition of goals in the logical view. In the use cases, the framework guided the translation of concerns and scenarios to decompositions of functionality and supporting components. The main problem with this *top-down* approach is that it requires the user to specify functionality and components rather early in the process. Substituting implementation details with *placeholders* might solve this problem in early phases of development.

Modularity

The logical, process, deployment and development viewpoints allow the user of the framework to decomposition complex models and functionality into compositions of smaller sub-models and sub-functionality. This addresses the *modularity* concern. The Production Cell use case demonstrated this modularity by separating control functionality from motion profile generation and I/O functionality.

Consistency

The framework provides an explicit set of correspondence rules with methods to check and resolve inconsistencies. In the use cases, the framework provided operations on views and correspondence rules to help in maintaining consistent views. Integrating the system's views in the iterative development cycle of the framework allows to incompatibilities to be resolve early and converge. Currently, no tooling is used to automatically evaluate consistency rules.

Reusability

As described in the *Reusing-phase*, the *reusability* concern is addressed by the framework because it provides its users with the ability to reuse components of an architecture description in different architectures. The architecture also separates computation and communication within MoCs from its development implementation which allows the re-usage of processes on different hardware architectures. The PowerGlove use case demonstrated how an existing system can be reused by recovering its architecture. In the Production Cell use case, the framework promoted reuse of software components through the separation of application-specific and generic functionality.

Extensibility

The framework allows extension of the architecture description by providing its user with the possibility to extend the architecture models and reuse existing components of these models. This addresses the *extensibility* concern. The Production Cell use case showed how the framework supports extending functionality of a system by adding additional safety concerns and updating the other views accordingly. The result was a working system that incorporates an *emergency signal* mechanism.

Testability

The *testability* concern is addressed by the many methods that the framework supplies for the testing of the system and parts of it in all phases of the development (modelling, simulation, prototyping, production). In the Pet Catcher use case, an approach to test the architecture through basic (co-)simulation was demonstrated. The Production Cell use case showed how to framework makes co-simulation and HIL possible to incrementally migrate from simulation to actual system. Though these approaches look promising, a range of problems persist that must be resolved to make it useable: the interface between co-simulated models requires custom wrappers and connectors which can be hard to write and error-prone. Furthermore, the performance of the Production Cell setup was currently very low. Meaningful simulation requires the use of appropriate models of the environment and interacting systems.

Depending on the CPS's context, modelling the environment and interacting subsystems can be a daunting task. Discrepancies between simulations and reality are another problem of this approach and the impact of these discrepancies can be hard to assess. In summary, a full system simulation is only feasible in a limited number of situations and when such a simulation is not possible, the user of the framework must resort to testing sub-systems instead and conducting integration tests on a prototype or actual physical plant.

Understandability

The models in the scenarios and deployment viewpoints correspond closely to the actual appearance of the system. Their explicitly defined relations with other viewpoints ensure that from every viewpoint the user of the architecture can understand how an architectural element of one of the views relates to the appearance of the actual system. This makes the architecture of the system understandable to the framework's users, addressing the *understandability* concern.

The PowerGlove use case demonstrated the process of architecture recovery to obtain an AD for the device. The question remains whether creating an AD of an existing system for the sake of understanding and documentation is worth the effort. This will probably depend on the preference and the experience of the user of the framework.

Simplicity

Although this work contains a lengthy elaboration of the rationale behind key decisions of the framework, the architectural approach itself is rather simple to apply and maintain as we showed in the previous chapter. The approach stimulates straight-forward architecture descriptions that relate closely to the goals that the system's stakeholders have in mind. Whether this actually results in a *simple* architecture remains a question that is particularly difficult to answer objectively. A possible way to determine this is to survey potential users of the AF to have them assess the *simplicity* of the resulting architectures.

7.4.2 Comparison with existing frameworks

AFs have been around for a long time and they have been proposed in different fields. Some of the more well-known ones are - introduced in section 1.6.2 - are:

- 4+1 Framework
- 5C Architecture
- CAFCR Framework

To compare these existing ones to the framework proposed in this document, they will be briefly discussed in the light of the domain concerns.

4+1 Framework

Kruchten [26] proposed with the *4+1 view model* one of the most popular software frameworks to date. He proposed 4 views:

Logical View Targets the functional requirements of the system, class diagrams form the central model to decompose the system into a set of key abstractions.

Process View The process view takes into account the organization of tasks, processes and their communication. The components are tasks and processes and they are connected with communication types, like messages or rendezvous.

Development View The development view focusses on the organization of software modules and subsystems. A layered hierarchy of components is a suitable model according to Kruchten.

Physical View Compares best to the deployment viewpoint of the AF. This viewpoint focusses on non-functional requirements like availability and reliability. Kruchten indicates that a range of notations is possible to model this view.

Scenarios Scenarios are used to discover architecture elements and to validate the resulting system. The typical notation of a scenario is a combination of the components of the logical view with the connectors of the process view.

The purpose of views of the 4+1 framework has much in common with that of the proposed AF. The later could even be seen as the 4+1 framework for CPSs but this would wrongly indicate a mere translation of the 4+1 framework to the CPS domain. The proposed AF was designed based on good practices in the worlds of systems, software and CPS design. Especially the process and deployment viewpoints of the proposed AF and the 4+1 framework are specific to respectively the CPS and the software domain. The 4+1 view model does not provide the unification of physical and cyber processes through the concept of MoCs, nor the insight of a process composition model which is essential to model the heterogeneity of CPSs. The physical view of the 4+1 view model describes the mapping of the software onto hardware. The comparable deployment viewpoint of the proposed AF focusses on the physical structure of the CPS by modelling it as a graph of cyber-nodes, physical components, sensors, actuators, communication media and external nodes. In the proposed AF, the logical view is not only used to manage the functionality of the system, but also to manage the functional and non-functional requirements of the system.

An advantage of the 4+1 view model is that each viewpoint is supported by a wide range of tooling and diagrams to visualize and analyse systems using the framework. The proposed AF is lacking such support.

5C Architecture

Lee, Bagheri and Kao [25] introduced the *5C Architecture* as a guideline for the implementation of CPSs in manufacturing systems. The architecture consists of 5 levels that target connection, conversion, cyber, cognition and configuration. For each level, possible algorithms and technologies are listed that could be used in realizing this level. The

5C architecture shares the concepts of cyber-models that mimic the behaviour of the related physical system.

The 5C architecture, however, does not specify the details, semantics and operations on each of the levels of the architecture. Consequently, it is hard to use the 5C architecture as a guideline for the development of (manufacturing) CPSs.

CAFCR Framework

The *Customer Objectives; Application; Functional; Conceptual; Realization* - or CAFCR - framework is developed by Muller [27]. It provides a framework for the creation of *products* and *systems*.

Customer objectives describes the system from a perspective of *key drivers* and *business models*. This viewpoint answers **what** the customer wants to achieve.

Application viewpoint describes the stakeholders and concerns and the system's context. The viewpoint answers **how** the customer realizes his goals.

Functional describes the *use cases*, a functional decomposition and non-functional requirements. The viewpoint answers **what** the system is.

Conceptual focusses on the concepts (stable) that implement the functionality of the system.

Realization focusses on the implementation details (unstable) of the product. Together, *Conceptual* and *Realization* answer the **how** questions of the system.

The customer objectives and application viewpoint are comparable to the stakeholders and concerns and the scenario viewpoint of the AF. The function viewpoint corresponds to the logical viewpoint of the AF and the conceptual and realization viewpoints correspond to the remaining three viewpoints.

The CAFCR architecture approaches system and product design from viewpoints that focus on different phases in the product realization. Especially the decomposition of the system implementation in *conceptual* and *realization* is different. Splitting up implementation in a stable and an unstable part is an appealing method to improve understandability, reusability and modularity. The logical viewpoint of the proposed AF features a layered hierarchy that shows some similarity to this method in the functional hierarchy.

7.4.3 Architecture Framework in the Context of Engineering Methods

An AF provides means to work with ADs in a specific domain. A framework describes both the types of elements of a corresponding architecture as well as the methods used to work with them. The AF proposed in this document describes a number of methods to work with the five viewpoints and the supporting architectural elements. Most of the methods and techniques used in the AF find their origin in existing development framework. Take for example iterative development; the development process describes by the V-model or Model-driven engineering. These techniques are no replacement for an AF but rather a companion to it: as a matter of fact, the user of the AF is recommended to use the tools and methods that they prefer to work with.

The added value of an AF is that it provides guidance during the creation of an AD in the form of viewpoints and supporting AEs, together with recommended practises for models and operations specific to the problem domain of the framework.

8 Conclusion

Designing CPSs is complex because of their heterogeneous and distributed nature and the great number of dependencies that arise from the stakeholders and their concerns. Possible applications of CPSs are promising but their realization is limited by the difficulty of designing and managing these complex systems. Reducing this complexity is key to enabling more promising CPS applications.

In an effort to cope with this complexity, this work introduces an Architecture Framework (AF) for Cyber-Physical System (CPS) that aims at improving design and management by helping in managing their complexity throughout their life cycle. The creation of the AFs is disguised in four research questions.

'What is a typical CPS?' boils down to a *base* CPS that consists of components that represent and interconnect the physical and the cyber world. The behaviour of the system is the result of distributed interaction between the two worlds. MoCs provide the abstractions that allow modelling individual parts of the system while their composition yields a model of the comprising system. Associated with these MoCs is the remaining difficulty of defining their interactions.

The aspects that improve the design and management of CPSs are selected as effectiveness, modularity, consistency, reusability, extensibility, testability, understandability and simplicity. By selecting these non-functional qualities, improvement can be defined without knowledge of the functionality of an actual CPS. These *domain concerns* relate to three groups of *domain stakeholders*.

The scenario viewpoint provides a situational overview and a way to take the result of other viewpoints together to validate the result. This viewpoint relies on specifying how the system should work. Ideally, a scenario is a representation of the actual situation and implemented in a computer testable way. However, though many tools exist to do this, some situations are difficult to model or simulate which limits the usefulness of the scenario.

The logical viewpoint aims at distilling what the goals of the system are and what elements are needed to support this goal. The resulting model is a decomposition of the system's primary objectives into a hierarchy of 'elements'. Adding functionality or changing requirements find an entry-point into the system in this hierarchy. The main difficulty remains to choose the right decomposition of components.

The process viewpoint approaches the system from a behavioural point of view. CPSs are by definition heterogeneous and the viewpoint takes that into account through the concept of Model of Computations (MoCs), providing a model-kind that allows for specifying the composition of MoCs. Modelling and design tools that allow composing heterogeneous MoCs are getting more common, allowing for a wider range of integration possibilities into project workflows. The interaction of MoCs requires attention as the rules of this interaction are ambiguous and may differ by convention or per tool while this interaction can significantly influence the resulting system.

The deployment viewpoint focusses on the connection of components of the physical and the cyber world. From this perspective, the connection and placement of physical components, cyber and external nodes, sensors, actuators and communication media can be considered. A consequence of separating behaviour and structure through the process and deployment view is their inherent dependence: linking of components yields alterations in their behaviour.

The development viewpoint focusses on the implementation of cyber-nodes: which connections, middleware and composition of processes give rise to corresponding cyber-nodes.

This selection of five viewpoints covers the general aspects of CPS design and the extensibility of the AF allows adding specific viewpoints to a system's AD.

By evaluating three use cases, the *successfulness* of the framework was assessed while at the same time, these use cases serve as application examples. The use cases showed by example how the framework copes with the eight central domain concerns. The proposed AF seems to benefit the process of design and management of CPSs in terms of the eight concerns. Yet, its application raises conceptual issues in itself like how to handle detailed functionality in an early design phase and how to integrate the design of physical components into the process. Besides these conceptual issues, practical issues will occur like: how to sufficiently model the environment and interacting systems and how to set up an appropriate simulation environment. Though this work showed that there are definitely benefits to using the proposed framework to design CPSs, there are also issues that need attention. In the end, it remains the architect's job to pick the right tools for the job and weigh the advantages and disadvantages of using a framework like that discussed in this work.

8.1 Future Research

Though the use cases showed the impact of the framework with respect to the domain concerns, the validation would benefit from applying the framework in an independent project. Additionally, future research should look further into comparison of the AF with existing frameworks, favourably through systematic analysis.

The consistency rules between views have been specified but no method for automatically checking and analysing them is proposed. As there is much information in the relation between the elements of views, future research should explore the automatic checking of consistency rules and the analysis of inconsistencies.

The TERRA-LUNA framework provides a development environment and software framework for the development of real-time software. Currently, the software supports the design of process views based on CSP models. Extending the software with the ability to compose heterogeneous MoCs makes TERRA-LUNA more suited for CPS design and development. The TERRA-LUNA architecture models provide a means to describe the cyber, media and interface parts of the deployment view but the physical part is lacking. Future research should point out the feasibility of adding physical components to these models.

Currently, no tooling for managing the hierarchical model of the logical view is used. Future research should look into an appropriate way to manage these models in a development environment and link the elements of the model and corresponding requirements to other parts of the system. This could improve the consistency of the architecture by allowing checking of correspondence rules and automatic evaluation of requirements. Principles of checking view consistency have been explored by Bhave et al. [50] and Rajhans et al. [17]. A similar problem holds for the development view model. The current model is represented by a layered diagram of development components that has to be translated to actual compilable code or binaries. Tooling could help in automating this process and ensuring consistency of the architecture.

References

[1] Edward Ashford Lee and Sanjit A Seshia. *Introduction to Embedded Systems*. A Cyber-Physical Systems Approach. MIT Press, Dec. 2016.

[2] Siddhartha Kumar Khaitan and James D McCalley. ‘Design Techniques and Applications of Cyberphysical Systems: A Survey’. In: *IEEE Systems Journal* 9.2 (May 2015), pp. 350–365.

[3] M Torngren et al. ‘Chapter 1 Characterization, Analysis, and Recommendations for Exploiting the Opportunities of Cyber-Physical Systems’. In: *Cyber-Physical Systems Foundations, Principles and Applications*. Elsevier, Jan. 2017, pp. 3–14.

[4] Amit Fisher et al. ‘Industrial cyber-Physical systems-iCyPhy’. In: *Complex Systems Design and Management - Proceedings of the 4th International Conference on Complex Systems Design and Management, CSD and M 2013*. International Business Machines, Armonk, United States. Jan. 2013, pp. 21–37.

[5] Wayne Wolf. ‘Cyber-physical Systems’. In: *Computer* 42.3 (Mar. 2009), pp. 88–89.

[6] J O Kephart and D M Chess. ‘The vision of autonomic computing’. In: *Computer* 36.1 (Jan. 2003), pp. 41–50.

[7] Levent Gurgen et al. ‘Self-aware cyber-physical systems and applications in smart buildings and cities’. In: *Proceedings -Design, Automation and Test in Europe, DATE*. CEA LETI, Grenoble, France. Oct. 2013, pp. 1149–1154.

[8] Mihnea Alexandru Moisescu et al. ‘Cyber physical systems based model-driven development for precision agriculture’. In: *Simulation Series*. University Politehnica of Bucharest, Bucuresti, Romania. Jan. 2017, pp. 57–67.

[9] K D Thoben, S Wiesner and T Wuest. ‘“Industrie 4.0” and Smart Manufacturing–A Review of Research Issues and Application Examples’. In: *Int J of Automation Technology . . .* (2017).

[10] Marija D Ilic, Le Xie and Usman A Khan. ‘Modeling future cyber-physical energy systems’. In: *Energy Society General Meeting*. IEEE, 2008, pp. 1–9.

[11] Yuchen Zhou, John Baras and Shige Wang. ‘Hardware Software Co-design for Automotive CPS using Architecture Analysis and Design Language’. In: *arXiv.org* (Mar. 2016). arXiv: `1603.05069v1 [cs.SE]`.

[12] Insup Lee et al. ‘Challenges and Research Directions in Medical Cyber-Physical Systems’. In: *Proceedings of the IEEE* 100.1 (Jan. 2012), pp. 75–90.

[13] Edward A Lee. ‘CPS foundations’. In: *Proceedings - Design Automation Conference*. UC Berkeley, Berkeley, United States. New York, New York, USA: ACM, Sept. 2010, pp. 737–742.

[14] Robert Bogue. ‘Towards the trillion sensors market’. In: *Sensor Review* 34.2 (Mar. 2014), pp. 137–142.

[15] Edward A Lee. ‘Cyber Physical Systems: Design Challenges’. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. IEEE, 2008, pp. 363–369.

[16] Patricia Derler, Edward A Lee and Alberto Sangiovanni Vincentelli. ‘Modeling Cyber-Physical Systems’. In: *Proceedings of the IEEE* 100.1 (2012), pp. 13–28.

[17] Akshay Rajhans et al. ‘Supporting Heterogeneity in Cyber-Physical Systems Architectures’. In: *IEEE Transactions on Automatic Control* 59.12 (2014), pp. 3178–3193.

[18] D L Parnas. ‘On the criteria to be used in decomposing systems into modules’. In: *Communications of the ACM* 15.12 (Dec. 1972), pp. 1053–1058.

[19] J Hill et al. ‘System architecture directions for networked sensors’. In: *Acm Sigplan Notices* 35.11 (Nov. 2000), pp. 93–104.

[20] Edsger W Dijkstra. ‘How Do We Tell Truths that Might Hurt?’ In: *Selected Writings on Computing: A personal Perspective*. New York, NY: Springer, New York, NY, 1982, pp. 129–131.

[21] Klaus-Dieter Thoben et al. ‘Considerations on a Lifecycle Model for Cyber-Physical System Platforms’. In: *Advances in Production Management Systems. Innovative and Knowledge-Based Production Management in a Global-Local World*. Berlin, Heidelberg: Springer, Berlin, Heidelberg, Sept. 2014, pp. 85–92.

[22] *ISO/IEC/IEEE International Standard - Systems and software engineering - Software life cycle processes*. IEEE. Nov. 2017.

[23] Dimitris Kiritsis. ‘Closed-loop PLM for intelligent products in the era of the Internet of things’. In: *Computer-Aided Design* 43.5 (May 2011), pp. 479–501.

[24] Software Systems Engineering Standards Committee of the IEEE Computer Society. *ISO/IEC/IEEE 42010:2011(E), Systems and software engineering — Architecture description*. Nov. 2011.

[25] Jay Lee, Behrad Bagheri and Hung-An Kao. ‘A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems’. In: *Manufacturing Letters* 3 (Jan. 2015), pp. 18–23.

[26] P Kruchten. ‘Reference: Title: Architectural Blueprints—The “4+ 1” View Model of Software Architecture’. In: *IEEE Software* (1995).

[27] G Muller. *Overview of CAFCR and Threads of Reasoning*. Tech. rep. Buskerud University College, Apr. 2004.

[28] R Hilliard. ‘Lessons from the unity of architecting’. In: *Software Engineering in the Systems Context*. mit.edu, 2015.

[29] Architecture Working Group of the Software. *Standards Department, IEEE: Recommended Practice for Architectural Description of Software Intensive Systems*. 2000.

[30] Joel Spolsky. *Joel on Software*. Berkeley, CA: Apress, Aug. 2004.

[31] Radha Poovendran. ‘Cyber-physical systems: Close encounters between two parallel worlds’. In: *Proceedings of the IEEE*. Aug. 2010, pp. 1363–1366.

[32] E W Kent and J S Albus. ‘Servoed world models as interfaces between robot control systems and sensory data’. In: *Robotica* 2.01 (1984), pp. 17–25.

[33] Ben van Lier and Teun Hardjono. ‘A Systems Theoretical Approach to Interoperability of Information’. In: *Systemic Practice and Action Research* 24.5 (Oct. 2011), pp. 479–497.
[34] Joseph Y Halpern and Yoram Moses. ‘Knowledge and Common Knowledge in a Distributed Environment’. In: *Journal of the ACM* 37.3 (July 1990), pp. 549–587.
[35] Jennifer Rowley. ‘The wisdom hierarchy: representations of the DIKW hierarchy’. In: *Journal of Information Science* 33.2 (2007), pp. 163–180.
[36] Martin Frick. ‘The knowledge pyramid: a critique of the DIKW hierarchy’. In: *Journal of Information Science* 35.2 (Apr. 2009), pp. 131–142.
[37] Nigel Goldenfeld and Leo P Kadanoff. ‘Simple lessons from complexity’. In: *Science* 284.5411 (Apr. 1999), pp. 87–89.
[38] Ilge Akkaya et al. ‘Systems Engineering for Industrial Cyber-Physical Systems Using Aspects’. In: *Proceedings of the IEEE* 104.5 (May 2016), pp. 997–1012.
[39] Jan C Willems. ‘The Behavioral Approach to open and interconnected systems’. In: *IEEE Control Systems Magazine* 27.6 (Dec. 2007), pp. 46–99.
[40] Edward A Lee and Alberto Sangiovanni-Vincentelli. ‘A framework for comparing models of computation’. In: *Ieee Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17.12 (Dec. 1998), pp. 1217–1229.
[41] Seyed Hosein Attarzadeh Niaki and Ingo Sander. ‘Co-simulation of embedded systems in a heterogeneous MoC-based modeling framework’. In: *SIES 2011 - 6th IEEE International Symposium on Industrial Embedded Systems, Conference Proceedings*. The Royal Institute of Technology (KTH), Stockholm, Sweden. IEEE, Aug. 2011, pp. 238–247.
[42] Antoon Goderis et al. ‘Heterogeneous composition of models of computation’. In: *Future Generation Computer Systems* 25.5 (May 2009), pp. 552–560.
[43] J Eker et al. ‘Taming heterogeneity - the Ptolemy approach’. In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 127–144.
[44] Seyed Hosein Attarzadeh-Niaki and Ingo Sander. ‘Integrating Functional Mock-up units into a formal heterogeneous system modeling framework’. In: *18th CSI International Symposium on Computer Architecture and Digital Systems, CADS 2015*. The Royal Institute of Technology (KTH), Stockholm, Sweden. IEEE, Jan. 2016, pp. 1–6.
[45] Hiren D Patel and Sandeep K Shukla. *SystemC Kernel extensions for heterogeneous system modeling: A framework for Multi-MoC modeling and simulation*. Boston: Kluwer Academic Publishers, Dec. 2005.
[46] Torsten Maehne, Alain Vachoux and Yusuf Leblebici. ‘Development of a bond graph based model of computation for SystemC-AMS’. In: *PRIME - 2008 PhD Research in Microelectronics and Electronics, Proceedings*. Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland. IEEE, Sept. 2008, pp. 77–80.
[47] A Bhave et al. ‘Augmenting software architectures with physical components’. In: (2010).
[48] M Athans. ‘The role and use of the stochastic linear-quadratic-Gaussian problem in control system design’. In: *IEEE Transactions on Automatic Control* 16.6 (Dec. 1971), pp. 529–552.
[49] Herman Bruyninckx. ‘Separation of Concerns: the “5Cs” — Levels of Complexity’. In: (Feb. 2011), pp. 1–5.
[50] Ajinkya Bhave et al. *View Consistency in Architectures for Cyber-Physical Systems*. IEEE Computer Society, Apr. 2011.
[51] B S Heck, L M Wills and G J Vachtsevanos. ‘Software Technology for Implementing Reusable, Distributed Control Systems’. In: *IEEE Control Systems Magazine* (Feb. 2003), pp. 21–35.
[52] Job Van Amerongen and Peter Breedveld. ‘Modelling of physical systems for the design and control of mechatronic systems’. In: *Annual Reviews in Control* 27 I.1 (Oct. 2003), pp. 87–117.
[53] Edsger W Dijkstra. ‘On the role of scientific thought.’ In: *Selected Writings on Computing: A personal Perspective*. New York, NY: Springer, New York, NY, 1982, pp. 60–66.
[54] Philippe Kruchten, Rafael Capilla and Juan Carlos Dueñas. ‘The Decision View’s Role in Software Architecture Practice’. In: *IEEE Software* 26.2 (), pp. 36–42.
[55] U Van Heesch, P Avgeriou and R Hilliard. ‘A documentation framework for architecture decisions’. In: *Journal of Systems and Software* 85.4 (Apr. 2012), pp. 795–820.
[56] J Tyree and A Akerman. ‘Architecture Decisions: Demystifying Architecture’. In: *IEEE Software* 22.2 (Mar. 2005), pp. 19–27.
[57] OMG. *Unified Modeling Language: Superstructure*. July 2004.
[58] Object Management Group. *OMG Systems Modeling Language*. May 2017.
[59] RobMoSys. *Separation of Levels and Separation of Concerns*. June 2017. URL: `http://robmosys.eu/wiki/general_principles:separation_of_levels_and_separation_of_concerns`.
[60] Dimitri Jeltsema and Jacquelien Scherpen. ‘Multidomain modeling of nonlinear networks and systems’. In: *IEEE Control Systems Magazine* 29.4 (), pp. 28–59.
[61] R N Taylor, A Van der Hoek Future of Software Engineering and 2007. ‘Software design and architecture the once and future focus of software engineering’. In: *FOSE’07*, pp. 226–243.
[62] Michael Nygard. *Documenting Architecture Decisions*. URL: `thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions`.
[63] Oliver Kopp, Anita Armbruster and Olaf Zimmermann. ‘Markdown Architectural Decision Records: Format and Tool Support ’. In: (Feb. 2018), pp. 1–8.
[64] Eamonn Keogh and Abdullah Mueen. ‘Curse of Dimensionality’. In: *Encyclopedia of Machine Learning and Data Mining*. Boston, MA: Springer, Boston, MA, 2017, pp. 314–315.
[65] Paul Hammant. *Testability and Cost of Change*. Nov. 2012. URL: `https://paulhammant.com/2012/11/01/testability-and-cost-of-change/`.
[66] Josien van den Noort et al. *Applications of the PowerGlove for Measurement of Finger Kinematics*. IEEE, 2014.
[67] Henk G Kortier, H Martin Schepers and Peter H Veltink. ‘Identification of object dynamics using hand worn motion and force sensors’. In: *Sensors (Switzerland)* 16.12 (Dec. 2016), p. 2006.
[68] J C van den Noort, H G Kortier and N van Beek. ‘Measuring 3D Hand and Finger Kinematics—A Comparison between Inertial Sensing and an Opto-Electronic Marker System’. In: *PLoS ONE* (2016).
[69] B Peerdeman et al. ‘A biomechanical model for the development of myoelectric hand prosthesis control systems’. In: *2010 32nd Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC 2010)*. IEEE, pp. 519–523.
[70] Marcel A Groothuis and Jan F Broenink. ‘HW/SW design space exploration on the production cell setup’. In: *Concurrent Systems Engineering Series*. University of Twente, Enschede, Netherlands. Dec. 2009, pp. 387–402.
[71] P J Vos. ‘Demonstrator combining ros/terra-luna’. In: (2015).
[72] L W van de Ridder. ‘Improvements to a Tool Chain for Model-Driven Design of Embedded Control Software’. PhD thesis. Enschede, Feb. 2018.

[73] Rich Hilliard. *Architecture Description of <Architecture Name> for <System of Interest>*. 2.2. Oct. 2014.

[74] T Blockwitz et al. ‘Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models’. In: *Proceedings* (2012).

[75] Peter Gorm Larsen et al. ‘Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project’. In: *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS, CPS Data 2016*. Aarhus Universitet, Aarhus, Denmark. IEEE, June 2016, pp. 1–6.

[76] C Andersson, J Akesson and C Führer. ‘PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface’. In: *Technical Report in . . .* (2016).

[77] Christian Andersson. ‘Methods and Tools for Co-Simulation of Dynamic Systems with the Functional Mock-up Interface’. In: *Doctoral Theses in Mathematical Sciences; (2016)* (2016).

[78] Design Automation Standards Committee of the IEEE Computer Society. *IEEE Std 1666-2011, IEEE Standard for Standard SystemC® Language Reference Manual*. Dec. 2011.

[79] Design Automation Standards Committee of the IEEE Computer Society. *IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual*. IEEE. 2016.

[80] Martin Krammer et al. ‘Standard Compliant Co-simulation Models for Verification of Automotive Embedded Systems’. In: *Languages, Design Methods, and Tools for Electronic System Design*. Cham: Springer International Publishing, 2016, pp. 29–47.

[81] Tom Lankhorst. *Control*. May 2018. URL: `https://github.com/tomlankhorst/control`.

[82] Gaël Guennebaud and Benoit Jacob. *Eigen*. 2010. URL: `http://eigen.tuxfamily.org/`.

Index

Appendices

A IEEE 42010 Standard Compliance

This appendix should show all requirements and how the AF addresses them

This appendix lists all requirements of a standard compliant AD and AF and shows how the proposed AF addresses them.

The following abbreviations hold:

S	Stakeholder
C	Concern
VP	Architecture Viewpoint
V	Architecture View
MK	Model Kind
M	Architecture Model
CR	Correspondence Rule
Co	Correspondence
R	Rationale and Architecture Decisions

The requirements of ADs are adapted from the AD template [73] and listed in table A.1. The requirements of an AF are a subset of these requirements, indicated by the absence of the *(AD only)* label.

Standard-compliance of the AF

Table A.2 indicates where the required components of a standard-compliant are located in the report.

Table A.1 Requirements of a IEEE 42010 compliant AD and AF, based on the AD template [73]

.

#	Requirement	Elements
1	**Stakeholders (S) and Concerns (C)**	
1.1	Identify and describe stakeholders	S
1.2	Consider predefined S groups See standard §5.3	S
1.3	Identify fundamental C's	C
1.4	Consider predefined C's See standard §5.3	C
1.5	Provide a S-C traceability table	S, C
2	**Viewpoints (VP)**	
2.1	Specify each VP used	VP
2.2	Each C must have at least one addressing VP	C, VP
2.3	Provide rationale for each VP	VP, R
2.4	Each VP has a representative name	VP
2.5	Describe the key features of each VP	VP
2.6	Document S's and C's that the VP targets	S, C, VP
2.7	Identify each MK used in the VP See standard §4.2.5, §5.5, §5.6	VP, MK
2.8	Describe each MK's conventions Refer to language, meta model, template or a combination	MK
2.9	Describe Operations on Views Construction, interpretation, analysis and implementation	VP, V
2.10	Document all CR's defined by the VP or M's	VP, M, CR
2.11	Provide examples or usage notes (optional)	VP
2.12	Provide sources According to standard §7	VP
3	**Views (V)** (AD only)	
3.1	Include a V for each VP	VP, V
3.2	Each V has a representative name	V
3.3	Provide one or more M's that adhere to VP	VP, V, M
3.4	M must address all C's framed by VP, for whole system	VP, M
3.5	Each M shall include versioning	M
3.6	Each M shall identify governing MK's	MK, M
3.7	V documents discrepancies between V and VP	VP, V
4	**Consistency and Correspondences**	
4.1	Record any known inconsistencies	CR, R
4.2	Provide consistency analysis of V's and M's (optional)	V, M, CR
4.3	Identify each Co, its CR's and participating AD elements	CR, Co
4.4	Identify each CR applying to the AD	CR
4.5	Check if each CR holds, record violations	CR
5	**Architecture Decisions and Rationale** (AD only)	
5.1	Provide evidence of consideration of alternatives and their rationale (optional)	R
5.2	Record key architecture decisions (optional)	R
5.3	Recorded decisions are structured See standard §5.8.2	R

Table A.2 Standard-compliance of the AF

	Stakeholders (S) and Concerns (C)											
Requirement	1.1	1.2	1.3	1.4	1.5							
Section	§3.1	§3.1	§3.2	§3.2	§3.2							
	Chapter 3 defines stakeholders and concerns of the AF.											
	Viewpoints (VP)											
Requirement	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10	2.11	2.12
Section	§4.1-§4.5									§4.6	Ch 6	
	The concerns, rationale, features and models are identified in the respective subsection of the viewpoints. Integrated examples are provided by the use cases.											
	Consistency and Correspondences											
Requirement	4.1	4.2	4.3	4.4	4.5							
Section	§4.6, Ch. 7		§4.6	*AD*	*AD*							
	Correspondence rules are proposed in section 4.6.											

B Co-simulation with FMI

The Functional Mock-up Interface (FMI) standard is a result of the MODELISAR project. FMI is a standard for co-simulation and model-exchange. FMI compatibility ensures that models comply to a standardized C API and provide model information through XML-files. FMI is an open standard managed by the Modelica Association. 95 tools on different platforms support FMI at the moment of writing. The packaged models are Functional Mock-up Units (FMUs) [74]. Co-simulation differs from model-exchange in that the solver algorithm is in the FMU, whereas with model-exchange, the solver of the FMI master simulates the model.

An example of a co-simulation setup with FMI is shown in figure B.1. This particular example shows the Co-simulation Orchestration Engine (COE) as FMI master with a dedicated manager application that controls the engine through HTTP. The Co-simulation Orchestration Engine (COE), part of the INTO-CPS project [75].

Although 95 tools support FMI this does not mean that they support all the FMI functionalities. For example, saving of model state and re-rolling it, is not supported in most tools. Not all models allow the retrieval of input-output derivatives which the FMI master can use for stabilization.

Let us show an example of the effect of co-simulation. We start with a simple dynamic model of a mass connected through a spring to a wall. The force on the mass is controllable and the distance from the wall is measurable. A suitably designed controller can make the mass follow a reference position. Figure B.2 shows this model implemented in 20-Sim.

20-Sim supports the generation of FMUs of sub-models. We generated FMUs for the Controller and the Plant. We compiled these FMUs with the same solver as we used in 20-Sim (fixed-step 1ms RK4) and executed a co-simulation using two tools: Ptolemy II and PyFMI. Ptolemy II was not able to resolve causality of the two sub-systems due to a dependency cycle. Including a delay of one solver iteration solves this problem. Figure B.3 shows the Ptolemy II configuration. Another tool, PyFMI, is a Python tool for FMI co-simulation based on Assimulo and a master algorithm for co-simulation [76]. PyFMI resolved the loop on its own.

We run the simulations with the settings listed in Table B.1. Figure B.4 shows

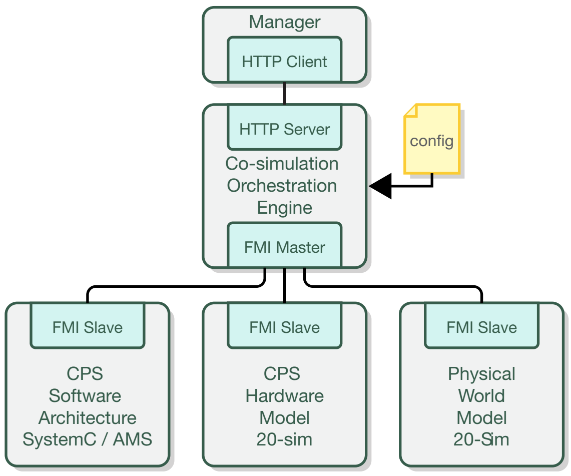

Manager
HTTP Client
HTTP Server
config
Co-simulation Orchestration Engine
FMI Master
FMI Slave
CPS Software Architecture SystemC / AMS
FMI Slave
CPS Hardware Model 20-sim
FMI Slave
Physical World Model 20-Sim

Figure B.1 Overview of the co-simulation environment

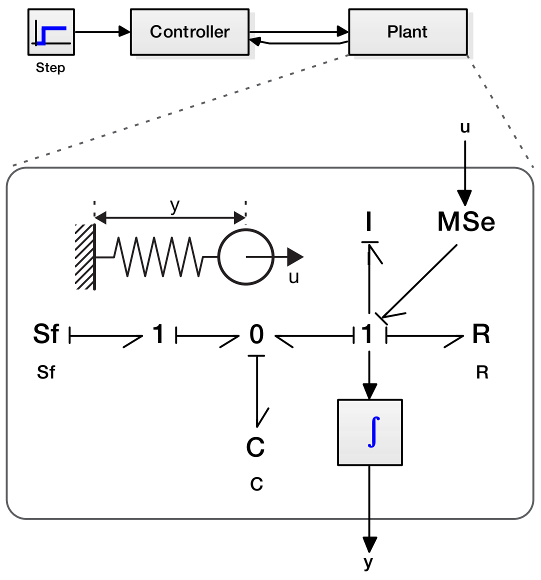


Figure B.2 20-Sim model of a controlled mass

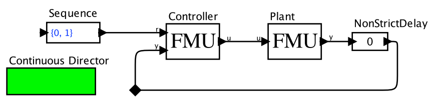


Figure B.3 Ptolemy II co-simulation

a graph of the time series of the output of the plant. The outcomes of the co-simulation environments differ from the reference simulation in 20-Sim. This is due to the delay that the programs introduced in the loop to solve the causality. In this case the system is stable but this deviation may as well result in instability of the closed loop.

The designer can decrease the effect of this delay by selecting a smaller time-step for both the master and the co-simulation models. As shown in Figure B.4, reducing the time-step of the master alone is not enough. Another option is to use iterative variable-step solvers. FMI supports this but not all simulators support exporting of variable-step solvers because functionality for restoring state is not included. FMI 2.0 provides an optional capability for FMUs that allows them to provide the system's Jacobian to the solver to improve the performance and stability of simulation [77].

Table B.1 Simulation environment and solver settings

tool	solver	timestep
20-Sim	Runge-Kutta 4	1ms
PyFMI	CVode	1ms
Ptolemy II	Runge-Kutta 4	10ms
Ptolemy II	Runge-Kutta 4	1ms
Ptolemy II	Runge-Kutta 4	0.1ms

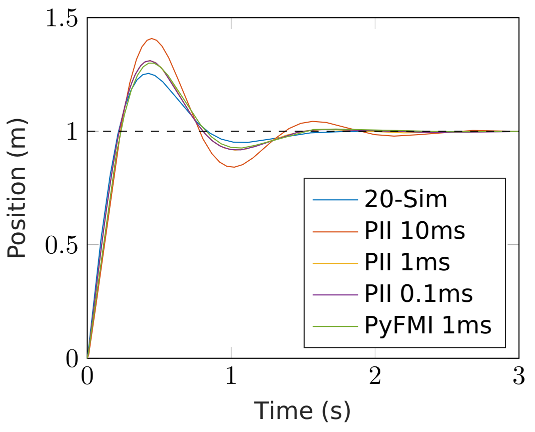


Figure B.4 Model Coupling with FMI Co-simulation

C Heterogeneous modelling with System-C

System-C is a suitable platform for heterogeneous systems modeling. SystemC is an extension to the C++ standard language that provides a simulation kernel for concurrent processes. The language specification is IEEE standardized [78]. The core language targets hardware and software simulation of embedded systems and hardware synthesis.

The main advantages of SystemC are:

- Standardized language and freely available reference implementation
- Different models of computation supported through extension
- Extension of well-known C++ language
- C++ enables both hardware and software modeling
- Big user community

Disadvantages are:

- Focus on hardware and software simulation, not on co-simulation with dynamic systems
- Hardware synthesis is mainly supported by commercial tools
- No built-in FMI compatibility

Standard SystemC is an extension to C++ that adds data-types such as four state logic (1/0/X/Z) and parallel digital inputs and outputs; port, interface and channel primitives; events sensitivity and notifications and a simulation interface for concurrent processes. SystemC targets to become the *default* modeling language for high-level embedded systems design. SystemC's default modeling abstraction is Register-Transfer Level (RTL). This level is the natural base-level of computers and supports discrete event Models of Computation (MoC) well.

SystemC Transaction-Level Modeling (TLM) is an addition to the RTL modeling provided by the language. The main advantages are *reduced modeling complexity* and *increased simulation performance*. TLM replaces register-level transactions by transactions of discrete messages.

SystemC Analog/Mixed-Signal Extension (AMS) is an extension to SystemC that provides more Models of Computation (MoCs): Timed Data Flow, Linear Signal Flow and Electrical Linear Networks. The extension enables modeling dynamic systems through differential equations. IEEE standardized this SystemC language extension [79].

Extending the SystemC kernel with custom MoCs is possible because the language is standardized and the reference implementation is open-source. Patel and Shukla [45] showed the possibility of extending the SystemC language with more models of computation. Maehne, Vachoux and Leblebici [46] developed a bond graph based model of computation.

SystemC has no standard support for co-simulation through FMI. Academics showed the feasibility of extending the kernel for enabling FMI co-simulation [80]. No standardized method is available at the time of writing.

D Architecture Decisions

D.1 Repository Structure

A typical Architecture Decisions repository for a CPS is tracked digitally and stays closely with the documentation, production diagrams and source code of the system.

The repositories of the systems in the use cases were Git repostories, added as submodules to the relevant system repostiories. This provides a tried and tested method for tracking and collaboration. The structure of the repository - using Markdown Architecture Decision Records - is basically:

```
README.md
decisions
|— 0000—use—markdown—architectural—decision—records.md
|— 0001—use—gazebo—simulation.md
```

The specific repositories used in the use cases utilized scripts to automatically convert *Markdown* files to LaTeX, HTML or other formats.

D.2 PowerGlove

Decisions regarding the recovery of the PowerGlove architecture.

ADR-PG-0001 Sample Rates

Context and Problem Statement

This decision concerns the sample and refresh rates of sensor data acquisition and hand pose reconstruction.

What must the sample rates of the acceleration and gyration sensors be? What must the refresh rate of the HPR be?

Considered Options

Constant rates in the range of 0 Hz to 1000 Hz.

Decision Outcome

Acceleration: 100 Hz, gyration: 200 Hz, magnetometer: 100 Hz.
Hand pose reconstruction: 20 Hz online, 50 Hz off-line, post processed.

- Finger and hand movement bandwidth is typically lower than 20 Hz
- 100 Hz acceleration and magnetometer sample rate and 200 Hz gyroscope sample rate have shown to be suitable rates in earlier work.
- The mentioned samples rates would require data bandwidth that are achievable with typical communication buses.
- On-line, the HPR should at least as an animation eye, 20 Hz is suitable for this.
- Off-line, the HPR should offer a more fluid result, 50 Hz is suitable for this.

ADR-PG-0002 Use FSM process model

Context and Problem Statement

The process model of the data-acquisition and IMU-reading processes are based on a FSM. On the actual device however, the implementation will involve interrupts that can alter the behaviour of the FSM.

What measures should be taken to account for this?

Considered Options

- Accept the discrepancy and handle it by ensuring its impact can be ignored
- Use a better process model that can handle the behaviour of interrupts
- Integrate interrupts in the FSM

Decision Outcome

Accept the abstraction and ensure its impact can be ignored.

Because it is possible to ensure that the impact can kept to acceptable proportions while the FSM description can stay in place, which is favourable because of its simplicity and clarity.

D.3 Production Cell

This is a collection of ADcs of the Production Cell setup.

ADR-PC-0000 Use Markdown Architectural Decision Records

Context and Problem Statement

Which format and structure should architecture decision records follow?

Considered Options

- MADR 2.1.0 - The Markdown Architectural Decision Records
- Michael Nygard's template - The first incarnation of the term "ADR"
- Sustainable Architectural Decisions - The Y-Statements

Decision Outcome

Chosen option: "MADR 2.1.0", because

- Implicit assumptions should be made explicit. Design documentation is important to enable people understanding the decisions later on. See also A rational design process: How and why to fake it.
- The MADR format is lean and fits our development style.
- The MADR structure is comprehensible and facilitates usage & maintenance.
- Markdown can be easily transformed to other document types using Pandoc

ADR-PC-0001 Use Gazebo for System Simulation

Context and Problem Statement

To focus and analyze the problem, align stakeholders' expectations, integrate and test the system, a simulation is favorable. Do we want to simulate the system? If so, what platform or framework should be chosen to do this simulation?

Considered Options

- No simulation
- Gazebo simulation framework
- 20-Sim simulation
- Unity Simulation

Decision Outcome

Chosen *Gazebo* because:

- Easy to use
- Superb integration with ROS and other robot tools
- Easier to model than 20-Sim
- More realistic physics engine compares to Unity

ADR-PC-0002 What Development Environment should be chosen

Context and Problem Statement

To develop the production cell, an IDE and set of tools can be very helpfull. What development environment is best?

Considered Options

- TERRA-LUNA
- Simulink + Embedded Coder
- 20-Sim + C-code generation
- Plain code editor

Decision Outcome

Decided *TERRA-LUNA* because:

- Open-source
- Supports graphical CSP
- TERRA Architecture Models can function as deployment models
- Drivers available for RAM-Stix single board computers
- LUNA provides a software stack that simplifies creation of the development view

ADR-PC-0003 What MoC is suitable as foundation

The MoC must guarantee real-time execution, provide overview during development and must be easy to interpret.

Considered Options

- Finite-State Machine
- Communicating Sequential Processes
- Synchronous Data-Flow

Decision Outcome

Chosen *Communicating Sequential Processes* because:

- Neatly integrates with TERRA environment
- Easy to interpret and to create
- Visualize-able through gCSP
- Can be checked for deadlock, live-lock

ADR-PC-0004 In what way should sub-systems collaborate?

What type of orchestration will serve the system best?

Considered Options

- Centralized Control (Master-Slaves)
- Decentralized Opportunistic

Decision Outcome

Decided *Decentralized Opportunistic* because:

- Robust to other components unresponsive or unreliable
- Problems are kept local, depending sub-systems will stop and wait in case of errors
- In line with the *self-dependence* concern

Accepting Downsides:

- Sub-system does not now as much about other sub-systems so control might be sub-optimal, affecting the *productive* concern.
- Still some kind of orchestration required for global messages (start/stop). Might be resolved in future through a communication scheme.

E.1 Process Diagrams

Process diagrams describe the behaviour of the subsystems. At the highest level the process diagrams of the Production Cell consist of graphical CSP diagrams. These diagrams contain *processes* which are described by state-machines.

Figure E.1 and Figure E.2 show the CSP diagrams of respectively subsystem 1 and subsystem 3. Central is a sequence of *acquire, control, output* at a fixed control rate. Also, there is a *system state in* and *system state out* process at an alternative rate. These processes are responsible for coordinating the system status and safety.

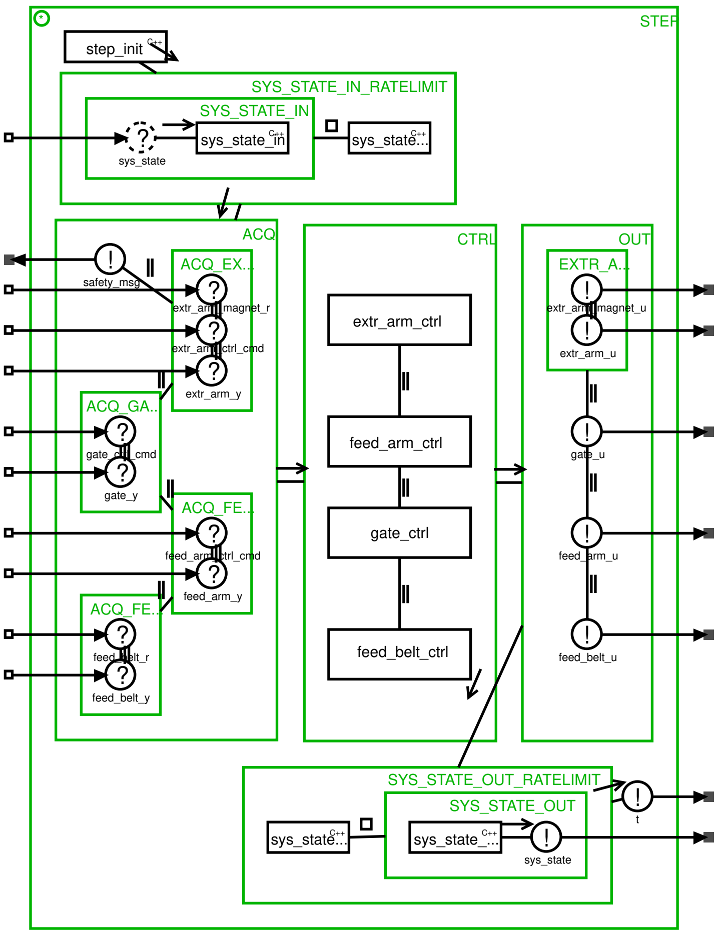


Figure E.1 CSP diagram of controller 1, note the typical *acquire, control, output* sequence and the *system state* in and output processes.

At a *higher* level, the subsystems have a *motion controller* that translates abrupt setpoints to smooth trajectories. Figure E.3 and Figure E.4 show the corresponding graphical CSP diagrams. The structure resembles that of a *acquire, control, output* sequence with trajectory-generators. The *control* stage is provided with a *homing guard* that ensures homing is done before normal operation.

At the *highest* level sequence controllers take care of the logical coordination of the system. The CSP diagrams represent again *acquire, control, output* sequences with an additional guard that ensures that ensures that the sequence starts when

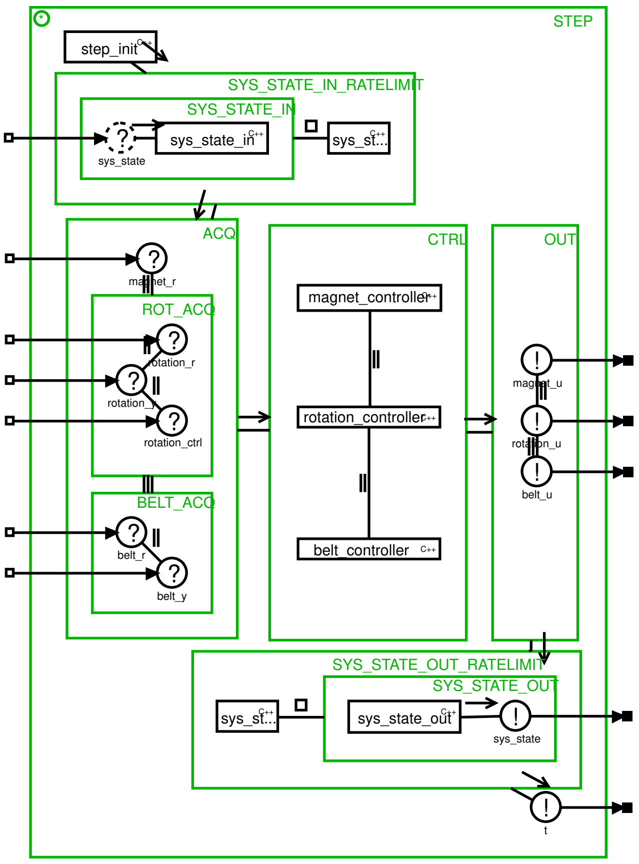


Figure E.2 CSP diagram of controller 3, comparable to number one but with fewer ports.

homing is done.

The homing sequence is represented by a relatively simple statechart as shown in figure E.7. The chart consists of four states linked by event-triggered or guarded transitions. Variables of concern are the reference frame (what direction is towards and what direction is away from home) and the homing velocity (timeout times position increment).

Figure E.8 shows the implementation of the rotation robot sequence controller. Because the arm of the robot can be at one place at a time, its position is a natural way to describe the main states of the robot. The *moving* states ensure that the machine finished a movement before accepting another. The E_RDY (ready) event fires when the settle-timer expires.

Figure E.9 and Figure E.10 show the statecharts of the feeder and the extraction arm of the molding subsystem. An interesting aspect of this subsystem is that the components share a physical space that they may not access concurrently or this leads to malfunctioning. As there is no overseeing sequence orchestration in the system, the subsystems have to coordinate access themselves. This *mutual exclusion* is implemented using the classic *Mutex* constructs. The regions of to mutual exclusion are the molder entrance and the exit gate:

Molder Entrace The feeder belt and feeder arm have to coordinate access to the entrance of the molder. Failing to ensure mutual exclusion may lead to damage to feeder arm or block.

Exit Gate The feeder arm, extraction arm and molder gate have to coordinate the opening of the gate. The gate must be and stay cloed when the feeder arm pushed a block and it must be open when the extraction arm picks a block. Failure to ensure the correct state of the gate results in possible damage to the molder or block.

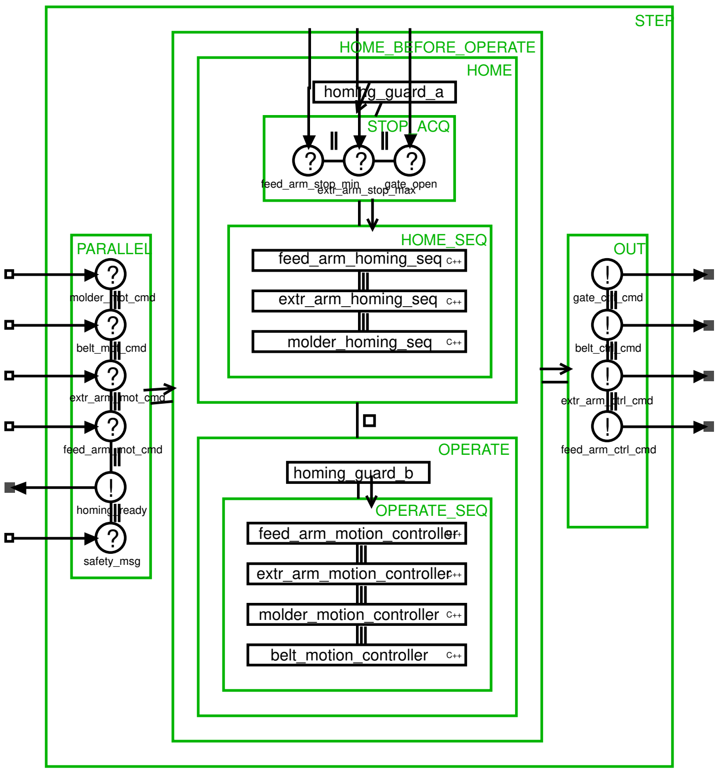


Figure E.3 The Motion Controller is a control loop with trajectory-generators combined with a homing guard.

E.2 Deployment Diagrams

The following diagrams show the deployment of sensors, actuators, nodes and communication. One diagram (Figure E.11) concerns the nodes and transducers related to the moulder part of the system, the other diagram (Figure E.12) shows those of the rotation-robot part of the system.

These diagrams originate from TERRA Architecture Models, part of the LUNA/TERRA software suite. Sensors are indicated by open black circles, actuators by solid black circles. Communication lines are indicated by black lines between filled and open small squares. The annotated black rectangles indicate nodes.

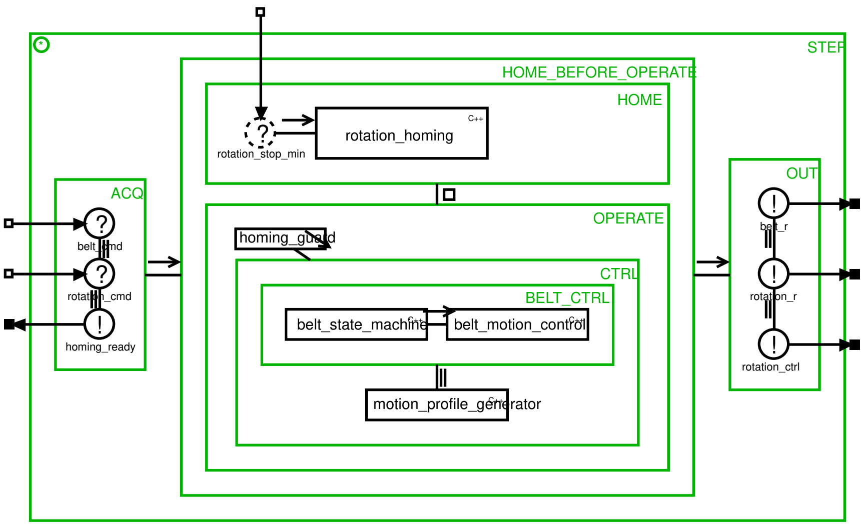


Figure E.4 Motion Controller 3 has a structure similar to Motion Controller 1

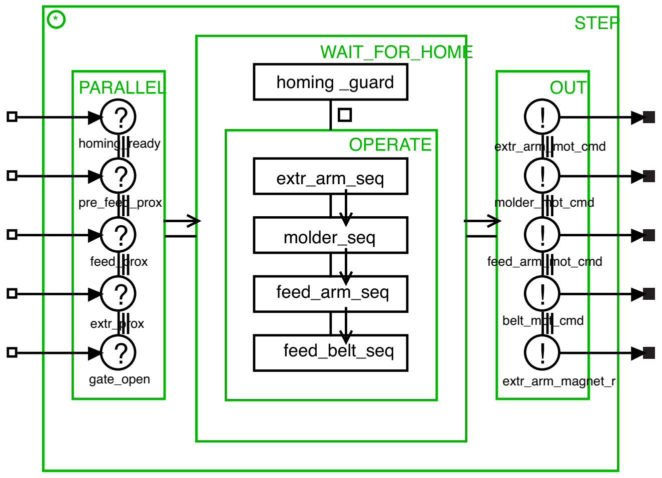


Figure E.5 Sequence controller CSP diagram. Note the straightforward implementation of read-compute-write. Most logic is state-related and thus handled in the corresponding state-machine processes.

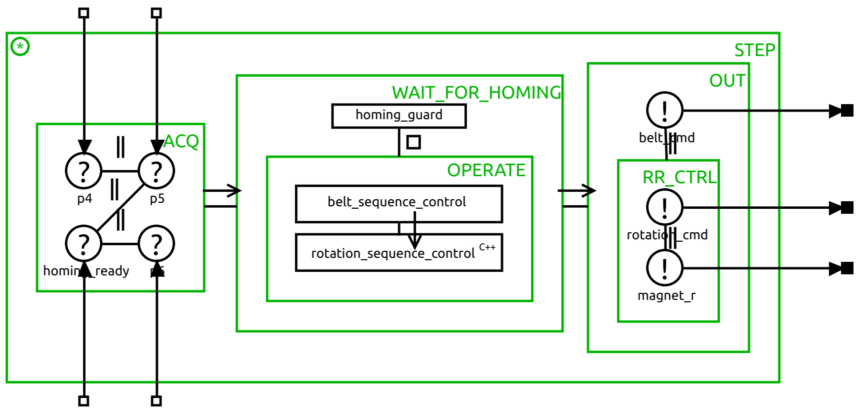


Figure E.6 The sequence controller 3 CSP diagram is comparable to that of sequence controller 1.

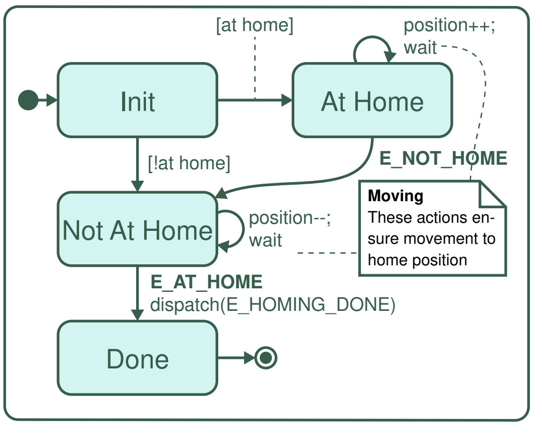


Figure E.7 Generic Homing Statechart Diagram. At start, determines whether already at home or not. Then, respectively moves away or towards home until at `Not At Home` the `E_AT_HOME` event is fired.

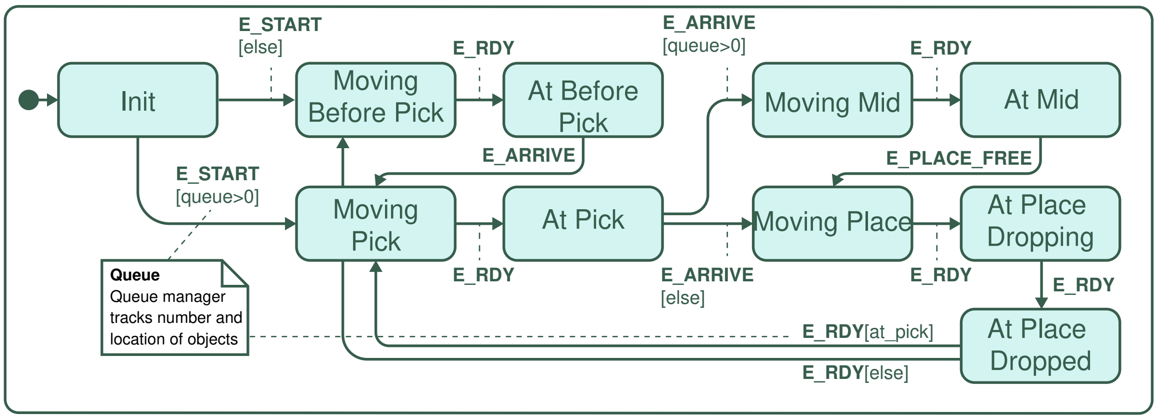


Figure E.8 Statechart of the sequence controller of the rotation robot. The chart takes into account the time required to reach a position with the *moving* states. A queue manager keeps track of objects.

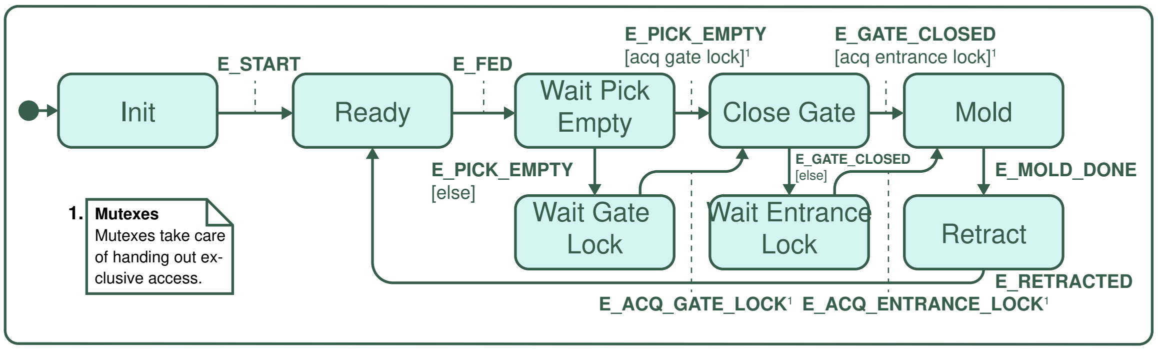


Figure E.9 Statechart of the sequence controller of the feed arm.

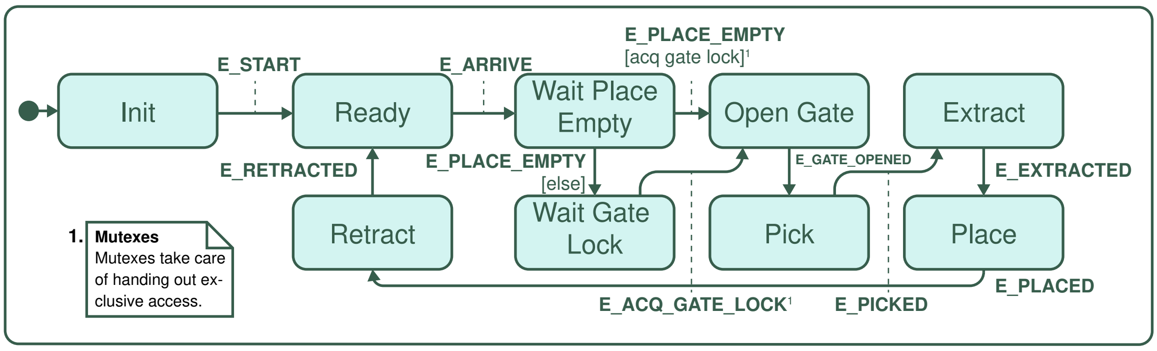


Figure E.10 Statechart of the sequence controller of the extraction arm.

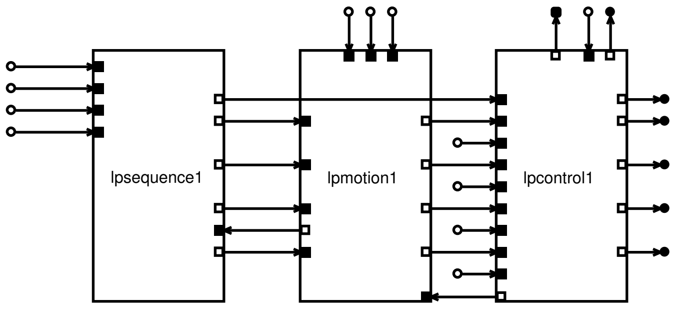


Figure E.11 Deployment diagram of Molder part

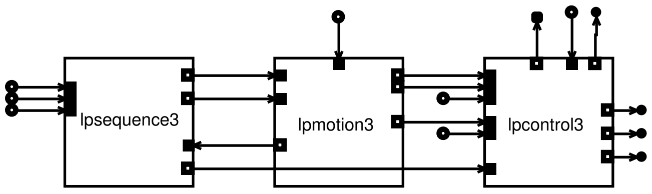


Figure E.12 Deployment diagram of Rotation Robot part

F Reusable Generic Control Algorithms in C++

An actuated physical system (*plant*) typically needs control to behave as desired. The subsystem responsible for controlling the plant is the *controller*. Design of the behaviour of the controller and consequently that of the plant is the concern of *control-theory*. Realization concerns translation of the behavioural description of a control-system (e.g. differential-equations) to an executable implementation.

The steps involved in the design and realization of a controller differ per situation and determining a satisfying strategy requires both insight in the control and implementation issues. There is initially significant effort involved in this process but it also poses an opportunity to create generic, reusable tools and software that make the process easier in later situations. This appendix demonstrates how such a general and reusable collection of code was developed in the context of the production-cell use case but aimed at general CPSs.

In CPSs this executable implementation typically boils down to software that utilizes digital control-algorithms to approximate the intended behaviour of the designed controller. DT LTI systems provide a wealth of possibilities for control design and because they are well-understood and suitable for generic computer implementation, they are the system type of choice.

The software is publicly available in a hosted Git repository [81].

F.1 LTI systems

CT LTI systems are typically expressed as transfer-functions in the frequency domain (with Laplace operator s) of the form:

$$\mathbf{Y}(s) = \mathbf{H}(s)\mathbf{U}(s) \tag{F.1}$$

In which $\boldsymbol{H}$ is possibly a Multiple Input Multiple Output (MIMO) transfer-function and $\mathbf{U}, \mathbf{Y}$ are respectively vectors of input and output signals.

The DT equivalent of (F.1) is a linear shift invariant function in the *z-domain*. To transform between CT LTI and DT systems, the *Bilinear transform* or *Tustin's method* can be used $\left(z = e^{sT} \approx \frac{1+sT/2}{1-sT/2}\right)$. The Bilinear transformation ensures that stable CT LTI systems map to stable DT systems.

To express the transfer function $\boldsymbol{H}$, a convenient representation is to be chosen. We choose two: the state-space representation and the *biquad* representation, having their own advantages and disadvantages.

F.1.1 State-Space Representation

The DT State-Space (SS) representation of an LTI system is:

$$\boldsymbol{x}z = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u} \tag{F.2}$$
$$\boldsymbol{y} = \boldsymbol{C}\boldsymbol{x}z + \boldsymbol{C}\boldsymbol{u} \tag{F.3}$$

Where the z operator denotes a time-shift of one-step forward. The dimensions of the matrices depend on the order $\mathrm{N_x}$ of the system and the number of out- and inputs $\mathrm{N_u}\mathrm{N_y}$:

- $\boldsymbol{A}$ is square $\mathrm{N_x}$
- $\boldsymbol{B}$ is $\mathrm{N_x}$ by $\mathrm{N_u}$
- $\boldsymbol{C}$ is $\mathrm{N_y}$ by $\mathrm{N_x}$
- $\boldsymbol{D}$ is $\mathrm{N_y}$ by $\mathrm{N_u}$

Characteristics of the SS representation include:

- Straight-forward MIMO implementation
- Poles, zeros, determinant computable from matrices
- Z-transform is bijective

F.1.2 Biquad Representation

Another representation of DT LTI systems is that of Biquads (or Second-Order-Sections). A biquad is particularly well suited for digital Single Input Single Output (SISO) filters. Higher-order filters can be represented by chaining biquads. It may be represented by the following transfer function:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \tag{F.4}$$

In which all coefficients are normalized by a_0. This yields the following difference equation for an input signal u and output signal y:

$$y = b_0 u + b_1 x z^{-1} + b_2 x z^{-2} - a_1 y z^{-1} - a_2 y z^2 {-1} \tag{F.5}$$

This is *direct form I* in which the output y is a function of two previous values of both u and y. This form can be rewritten to *transposed direct form II*:

$$s_2 = b_2 x - a_2 y \tag{F.6}$$
$$s_1 = s_2 z^{-1} + b_1 x - a_1 y \tag{F.7}$$
$$y = b_0 x + s_1 z^{-1} \tag{F.8}$$

In which the output y is a function of just two intermediate variables.

Characteristics of the biquad representation include:

- Requires just two state-variables per section
- Suited for SISO systems
- Straight-forward digital implementation

F.2 Filters and Controllers

LTI transfer functions can represent the behaviour of a range of systems. The systems of interest in the context of control-design are that of filters and controllers.

Filters find applications in noise-reduction and signal conditioning.

Controllers aim at regulating a plant. One of the most-used and general-purpose controller-families is the Proportional Integral Derivative (PID) controller and its relatives. Simple mass-damper-spring-like systems can be accurately controlled using these controllers.

More advanced control-architectures are built around pole-placement such as Linear-Quadratic Regulators (LQRs). These architectures require knowledge of the full-state which is generally not trivially available. Consequently, an observer needs to be added to the control loop. State-space models can be used to digitally implement an observer and controller and are therefore very useful in the general design of systems.

F.3 Implementation Considerations

This section discusses the implementation details of the filter and control functionality discussed above.

F.3.1 Computer Arithmetic

Filters and controllers typically involve real numbers in $\mathbb{R}$. In digital computers, these numbers are represented by data-types such as floating-point and fixed point. Different types have different characteristics in terms of performance, range, precision, storage size and performance. To allow the developer to choose an appropriate data-type, the control functions are *templated*: a C++ language feature allowing users of code to dictate its data-types. A single-precision floating-point (`float`) is the default-data type which suffices for simple control and filter tasks while allowing hardware-floating point processing and vectorization on most targets.

F.3.2 Biquad Implementation

A single biquad section is implemented as a Transposed Direct Form II with user-preferred arithmetic type.

Listing F.1 Execution of a single biquad step.

```
T step(T x){%
        T y;
        y     = x * B[0]            + wz[0];
        wz[0] = x * B[1] - A[0] * y + wz[1];
        wz[1] = x * B[2] - A[1] * y;
        return y;
}
```

One step of a single biquad requires 5 multiplications, 4 additions and access to two state-variables. The compiler will be made responsible for applying vectorization when possible.

F.3.3 **State-space Implementation**

Digital implementation of arbitrary order SSs required matrix algebra functionality. This functionality is much used, generic and suitable for computer implementation. Consequently, open-source libraries are available that provide these matrix algebra functions for us. The nature of matrix operations allows for parallelisation of operations, also known as vectorization or *Single Instruction Multiple Data*. To exploit this performance potential and reduce development-time, the open-source, templated, tried and testes *Eigen 3* [82] library is used.

Templates are used to allow the user of the code to specify the data-types and number of states N_x, inputs N_u and outputs N_y. Eigen's comma-initialization syntax allows straight-forward initialization of matrices and its operator overloading enables semantic and easy to understand code. The core of the SS now boils down to:

Listing F.2 Execution of a single step. Ty and Tu are types representing Eigen vectors of appropriate arithmetic type.

```
Ty step(Tu u){%
    x = A*x + B*u;
    y = C*x + D*u;
    return y;
}
```

Advantages of a state-space implementation include that the dynamics of the system can be more easily understood from the system matrices and that the full-state is easily available.

F.3.4 **PID(F) Control**

PID(F) controllers are implemented as single biquad filters. The control algorithms allow construction of a PID(F) controller by providing K_p, τ_i, τ_d, N and T_s parameters directly to the constructor. Furthermore, a PI controller is a PID with $\tau_d = 0$ and $N = \infty$, a PD controller is a PID with $\tau_i = \infty$ and a P controller is a simple multiplication. The corresponding biquad coefficients are calculated at compile-time, resulting in zero run-time overhead.

The formulas to calculate b_0 to b_2 and a_1 and a_2 are:

$$b_0 = \frac{K_p\,\left(4\,\tau_d\,\tau_i + 2\,\tau_d\,T_s + N\,{T_s}^2 + 4\,N\,\tau_d\,\tau_i + 2\,N\,\tau_i\,T_s\right)}{2\,\tau_i\,\left(2\,\tau_d + N\,T_s\right)} \tag{F.9}$$

$$b_1 = -\frac{K_p\,\left(-N\,{T_s}^2 + 4\,\tau_d\,\tau_i + 4\,N\,\tau_d\,\tau_i\right)}{\tau_i\,\left(2\,\tau_d + N\,T_s\right)} \tag{F.10}$$

$$b_2 = \frac{K_p\,\left(4\,\tau_d\,\tau_i - 2\,\tau_d\,T_s + N\,{T_s}^2 + 4\,N\,\tau_d\,\tau_i - 2\,N\,\tau_i\,T_s\right)}{2\,\tau_i\,\left(2\,\tau_d + N\,T_s\right)} \tag{F.11}$$

$$a_1 = -\frac{4\,\tau_d\,z}{2\,\tau_d + N\,T_s} \tag{F.12}$$

$$a_2 = \frac{2\,\tau_d - N\,T_s}{2\,\tau_d + N\,T_s} \tag{F.13}$$

These formulas are entered as `constexpr` functions in the C++ PID class.

Down-side of a biquad implementation is the lack of anti-windup measures for the integration action. If such a measure is required, a state-space implementation might be preferred.

G Motion Profiles

Appropriate motion profiles allow a fluent transition from an ideal position controlled system to an actual servoed plant. Typically, polynomial profiles are preferred because of their straight-forward differentiability and mathematical properties. A polynomial motion profile of arbitrary order and its derivative are given by:

$$M_N(x) = a_0 + a_1 x + a_2 x^2 + \dots a_N x^N \tag{G.1}$$

$$\frac{d}{dx} M_N(x) = a_1 + 2a_2 x + \dots N a_N x^{N-1} \tag{G.2}$$

The coefficients a_n can be solved when the equations are properly constrained. Typically, the begin and end position, position derivative, etcȧre constrained. Now finding the appropriate coefficients is a matter of solving:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^N \\ 1 & x_f & x_f^2 & \dots & x_f^N \\ 0 & 1 & 2x_0 & \dots & N x_0^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \ddots \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} M_N(x_0) \\ M_N(x_f) \\ \frac{d}{dx} M_N(x_0) \\ \vdots \\ \frac{d^N}{dx^N} M_N(x_f) \end{bmatrix} \tag{G.3}$$

In which x_0 and x_f are respectively the initial and final values of x. Now the vector A can be solved from the linear equation. The differentiability of an n-th order polynomial profile ensures that no jumps occur in the first n derivatives of the profile, reducing acceleration, jerk and so forth to reduce the control error of a plant.