RobMoSys

H2020—ICT—732410

RobMoSys

COMPOSABLE MODELS AND SOFTWARE FOR ROBOTICS SYSTEMS

DELIVERABLE 6.4: REPORTS OF EXPERTS WORKSHOPS

Sara Tucci (CEA)

THIS PROJECT HAS RECEIVED FUNDING FROM THE *EUROPEAN UNION'S HORIZON 2020 RESEARCH AND INNOVATION PROGRAMME* UNDER GRANT AGREEMENT NO. 732410

Project acronym: RobMoSys

Project full title: Composable Models and Software for Robotics Systems

Work Package: [WP 6]

Document number: 6.4

Document title: Reports of experts workshops

Version: [1.0]

*Due date: [*December 31th, 2017]

Delivery date: 31.12.2017

Nature: [Report (R)]

Dissemination level: [Public (PU)]

Editor: [Sara Tucci (CEA)]

Author(s): Sara Tucci (CEA)

Reviewer: [Dennis Stampfer (HSU)]

Executive Summary

This report summarizes the content of the first Expert Workshop held in Frankfurt the 7th-8th of February 2017.

This is the first workshop of a series of Expert Workshops we set along the project lifetime to gather all the possible insights and knowledge (mainly from near communities and industrial representatives) to (i) evaluate best-practices established in near and mature domains and (ii) identify current showstoppers that could arise in the robotics domain. This understanding is necessary to make sure that Open Calls will be prepared to provide concrete answers to the community, to finally overcome identified showstoppers and secure broad adoption.

Content

1 Introduction

This is the first workshop of a series of Expert Workshops we set along the project lifetime to gather all the possible insights and knowledge (mainly from near communities and industrial representatives) to (i) evaluate best-practices established in near and mature domains and (ii) identify current showstoppers that could arise in the robotics domain. This understanding is necessary to make sure that Open Calls will be prepared to provide concrete answers to the community, to finally overcome identified showstoppers and secure broad adoption.

For this first workshop, the Consortium invited experts, from relevant related domains, with a strong scientific background in the design of complex and critical systems, namely:

- Arne Haman (Bosch) from automotive and embedded systems
- Jan Broenink (University of Twente) mechatronic/cyber physical systems – hybrid simulation
- Saddek Bensalem (Verimag) cyber physical systems – formal methods
- Jurgen Bock (Kuka) Industrie 4.0 – ontologies, semantic technologies

To prepare the workshop, each Expert discussed in an individual teleconference with the Consortium the general objectives of RobMoSys and the particular mission we were about to give to them. In order to set the context, the Consortium presented the following questionnaire:

"What is the aim of RobMoSys? RobMoSys envisions an integrated approach built on top of the current code-centric robotic platforms, by applying model-driven methods and tools. RobMoSys will enable the management of the interfaces between different robotics-related domains in an efficient and systematic way according to each system's needs. RobMoSys aims to establish Quality-of-Service properties, enabling a composition-oriented approach while preserving modularity. RobMoSys will drive the non-competitive part of building a professional quality ecosystem by encouraging the community involvement. RobMoSys will elaborate many of the common robot functionalities based on broad involvement of the community via two Open Calls. These are the topics we would like to hear your opinion on:

- How do you deal with composition and which are your priorities?
- How do you make sure that different vocabularies in connected components semantically match?
- How do you manage the link between composition and certification?
- How do you deal with quality-of-service properties (extra-functional properties) and how do you make sure quality-of-service is right?
- How do you assess good practices and what kind of metrics do you apply?
- What is still missing?
- In the different steps of the process, which one do you think is the hardest part & how would you solve it or absolutely not solve it?"

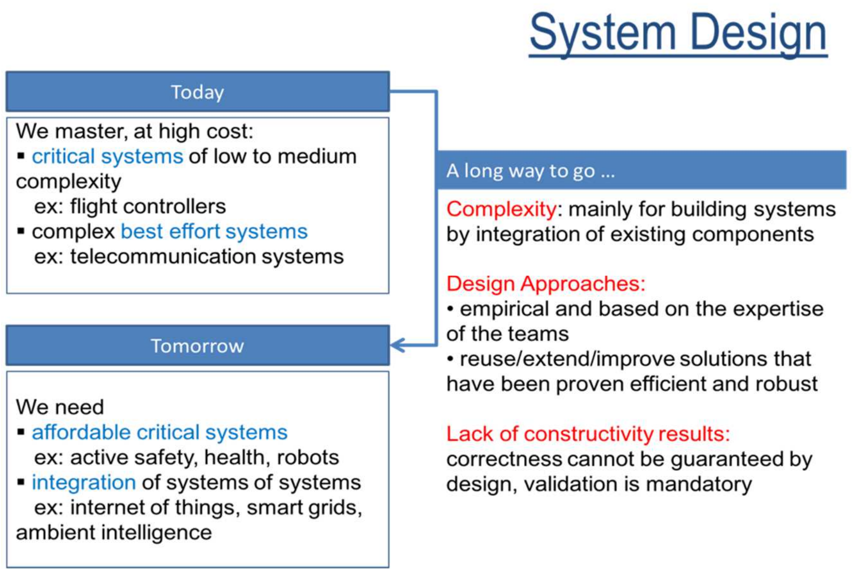
In the first part of this report (Experts Contributions) we summarize the contribution of each expert: the content of the presentation the expert made the first day. The second part (Current design methodologies assessment) presents the output of the second day: each expert was asked to fill a table pointing out several aspects and pain-points of presented design approaches. The third part of the report (Synthesis on recommendations for RobMoSys) summarizes the general recommendations of the experts.

2 Experts Contributions

2.1 Saddek Bensalem: a formal framework for system design

Saddek Bensalem presented principles and concepts related to rigorous system design. Saddek pointed out that reactive systems are increasingly important in modern computing systems: embedded systems, cyber-physical systems, mobile systems, web-services. They are hard to design due to unpredictable and subtle interactions with the environment, emergent behaviors, etc. Robots are a class of Cyber-Physical Systems.

System design is facing several difficulties, mainly due to our inability to predict the behavior of an application software running on a given platform. Other difficulties stem from current design approaches, often empirical and based on expertise and experience of design teams. Naturally, designers attempt to solve new problems by reusing, extending and improving existing solutions proven to be efficient and robust. This favors component reuse and avoids re-inventing and re-discovering designs. Nevertheless, on a longer-term perspective, this may also be counter-productive: designers are not always able to adapt in a satisfactory manner to new requirements. Moreover, they a priori exclude better solutions simply because they do not fit their know-how.



System Design

Today

We master, at high cost:
- critical systems of low to medium complexity
  ex: flight controllers
- complex best effort systems
  ex: telecommunication systems

Tomorrow

We need
- affordable critical systems
  ex: active safety, health, robots
- integration of systems of systems
  ex: internet of things, smart grids, ambient intelligence

A long way to go …

Complexity: mainly for building systems by integration of existing components

Design Approaches:
• empirical and based on the expertise of the teams
• reuse/extend/improve solutions that have been proven efficient and robust

Lack of constructivity results: correctness cannot be guaranteed by design, validation is mandatory

Limitations of V-like models of traditional Systems Engineering processes can also be observed in this context. Indeed, V-like models:

1. assume that all the system requirements are initially known, can be clearly formulated and understood.

2. assume that system development is top-down from a set of requirements. Nonetheless, systems are never designed from scratch; they are built by incrementally modifying existing systems and by component reuse.

3. consider that global system requirements can be broken down into requirements satisfied by system components. Furthermore, it implicitly assumes a compositionality principle: if components are proven correct with respect to their individual requirements, then correctness of the whole system can be inferred from correctness of its components.

4. rely mainly on correctness-by-checking (verification or testing)

To overcome limitations of current approaches, a novel rigorous approach for system design is then needed, promoting the following principles:

- **Separation of Concerns**
- **Component-based approach**
- **Semantic Coherence**
- **Correct-by-construction**;

while rising three Grand Challenges:

- **Marrying Physicality and Computation**. We need theory and models encompassing continuous and discrete dynamics to predict the global behavior of a system interacting with its physical environment. The development of application software and its implementation must take into account constraints from: the physical resources and the physical environment of the system.

- **Component-based Design**. We need theory, models and tools for the cost-effective building of complex systems by assembling heterogeneous components

- **Adaptivity**. Systems must provide a service meeting given requirements in interaction with uncertain environments. Uncertainty can be characterized as the difference between average and extreme system behavior. Non-determinism of physical environments increases uncertainty.

and meeting the following objectives:

- **Productivity**. This can be achieved by system design flows providing ***high level domain-specific languages*** for ease of expression, allowing reuse of components and the development of component-based solutions, integrating tools for programming, validation and code generation.

- **Performance**. The design flow must allow the satisfaction of **extra-functional properties** regarding optimal resource management. This means that resources such as memory, time and energy are first class concepts encompassed by formal models. Moreover, it should be possible to analyze and evaluate efficiency in using resources as early as possible along the design flow. Design space exploration should be promoted to resolve choices such as reducing parallelism (through mapping on the same processor), reducing non-determinism (through scheduling), fixing parameters (quality, frequency, voltage).

- **Correctness**. This means that the designed system meets its specifications. Ensuring correctness requires that the design flow relies on models with ***well-defined semantics***. The models should consistently encompass system description at different levels of abstraction from application software to its implementation. Correctness can be achieved by application of verification techniques.

To meet these objectives **model-based and component-based design should be merged** in a uniform **formal framework** with the following characteristics:

- Adopt a ***model-based design***. Model-based design means that software and system descriptions used along the design flow are based on a single semantic model. This is essential for maintaining the overall coherency of the flow by guaranteeing that a description at step n meets essential properties of a description at step n - 1. This means in particular that the semantic model is expressive enough to directly encompass various types of component heterogeneity arising along the design flow.

- Adopt a ***component-based approach***. Component-based design promotes composability and compositionality principles. The key issue is how to build systems from a set of given atomic components (behavior) that meet a given property. This is a hard problem. Nevertheless, it is important to have a framework for tackling this problem and decomposing it into simpler problems. That is to have a construction methodology.

Build a component C satisfying a given property P, from

1. Co a set of atomic components modeling behavior
2. GL ={gl1, ..., gli, ...} a set of glue operators on component

Glue operators:

- model mechanisms used for communication and control such as protocols, controllers, buses.
- restrict the behavior of their arguments.

The formal framework should offer a minimal set of constructs and principles for guaranteeing **correctness by construction**, *such as decomposition and flattening* **as depicted** in Figure 1.

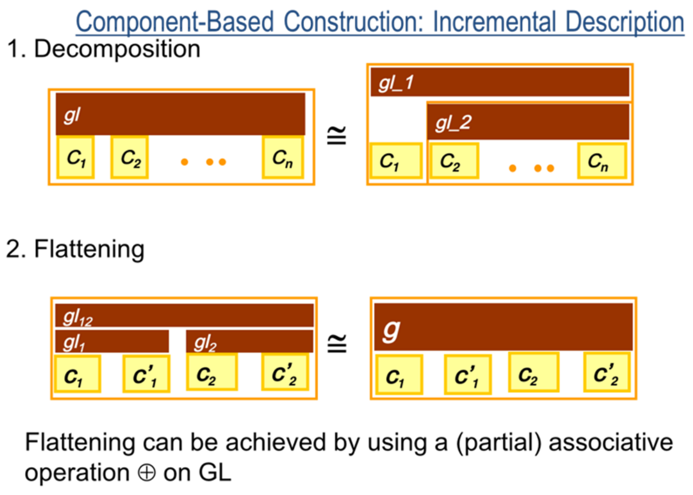

Component-Based Construction: Incremental Description
1. Decomposition
gl
C_1 C_2 C_n
≅
gl_1
gl_2
2. Flattening
gl_{12}
gl_1
gl_2
C_1 C'_1 C_2 C'_2
g
Flattening can be achieved by using a (partial) associative operation ⊕ on GL

Figure 1: Decomposition and Flattening

The framework enjoys a generalization of associativity. Any n-ary glue operator is the composition of binary glue operators - This is very important for systems with dynamically changing structure. Dually, hierarchically structured components can be flattened – single glue operator applied to the atomic components. To achieve flattening some composition operation on glue is needed.

Component-based construction is based on two important properties, namely **composability** and **compositionality**. Composability is about composing components without breaking their properties after composition. Composability guarantees preservation of a component property across integration.

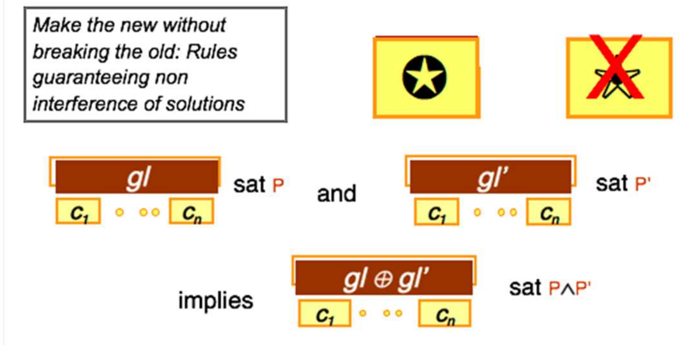

Make the new without breaking the old: Rules guaranteeing non interference of solutions
gl
C_1 ... C_n
sat P
and
gl'
sat P'
implies
gl ⊕ gl'
sat P∧P'

Figure 2: Composability

Compositionality allows deduction of the composed global properties from its component properties; this property enables correctness- by-construction.

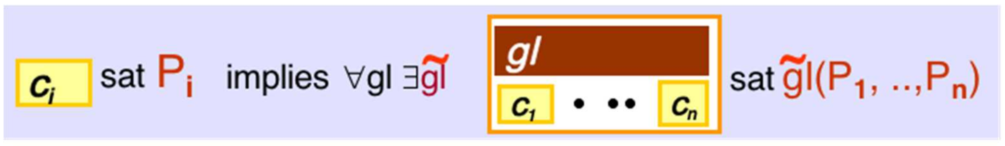

C_i sat P_i implies $\forall gl\ \exists \tilde{gl}$
gl
C_1 ... C_n
sat $\tilde{gl}(P_1, .., P_n)$

Figure 3: Compositionality

To make an example Figure 4 shows two kind of compositions. The first one is a composition in which the composed components satisfy a certain property (no deadlock), and the property is preserved at the level of the composite component thanks to a proper operator. In the second composition, components do not satisfy any deadlock-free property, but the composite satisfies a Mutex property thanks to a Mutual Exclusion operator.

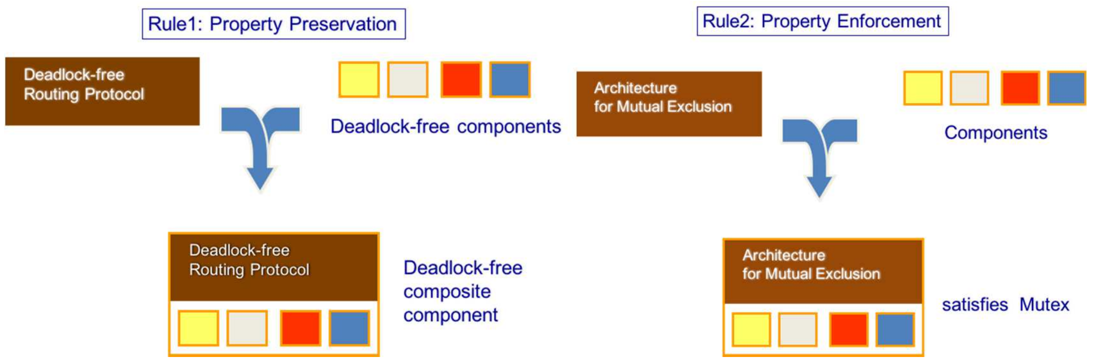

Rule1: Property Preservation
Deadlock-free Routing Protocol
Deadlock-free components
Deadlock-free Routing Protocol
Deadlock-free composite component
Rule2: Property Enforcement
Architecture for Mutual Exclusion
Components
Architecture for Mutual Exclusion
satisfies Mutex

Figure 4: Examples of Compositions and Related Properties

- The formal framework should also be expressive enough to encompass ***heterogeneity of execution*** (synchronous and asynchronous components); interaction (function call, broadcast, rendez-vous); abstraction levels (hardware, middleware, application software).

- The formal framework should as well provide ***automated support*** for efficient implementation on given platforms and automated support for validation and performance analysis following the design flow suggested in Figure 5.

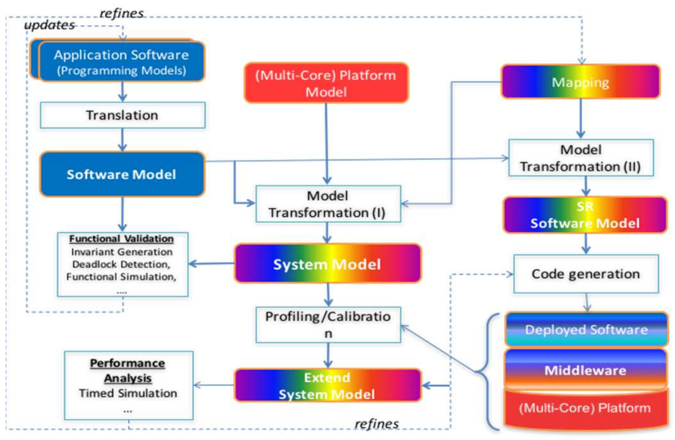


Figure 5: Design Flow

2.2 Arne Hamann: Essential Analysis and QoS management

Arne Hamann pointed out that ***model driven methods are key to boost design efficiency and confidence***. What makes model-driven methods fundamental is their suitability to compose functionalities on ***system level and then derive/predict system-level properties***. However, finding the right models & abstractions is extremely hard and requires in-depth domain knowledge. Many bad examples are out there, which lead to the bad reputation of model based methods, like for instance using UML for modelling software. UML models often either lack a meaningful level of abstraction or lack clear semantics. UML models are, therefore, perceived to be only of little help by many software engineers.

2.2.1 Essential Analysis for Functional Domains

In order to deal with ***composition*** and issues on an underlying ontology supporting meaningful semantic connection among components, the suggestion here is to refer to morphological and essential system analysis[1]. These tools give help identifying right abstractions and concepts with respect to a given functional domain.

Essential Analysis (Figure 7) systematically ***decomposes the overall problem space according to discrete "situations" in the system context***. The objective is to identify sub-problems called **system *modes*** that can be solved independently. The obtained system modes can be used as blueprint to structure the implementation in later development stages. In the final implementation this automatically leads to the separation of control flow and the data flow. The decomposition of the problem space and the identification of system modes is based on a light-weight formalisation of system knowledge, i.e. a compact and unambiguous specification. This approach allows to check two fundamental properties during system design:

- **Completeness**: all possible states in problem space have been analysed
- **Consistency**: each part of the problem space belongs exactly to one system mode

Obviously, the decomposition of the functional domain can be carried out only through a dialogue between the System Expert that has a specific domain knowledge and the Methodologist that has a specific knowledge about the method. Essential analysis (including consistency and completeness checks) can be performed thanks to a tool proposed by ETAS[2].

---

[1] https://en.wikipedia.org/wiki/Morphological_analysis_%28problem-solving%29
Essential System Analysis - Stephen M. McMenamin, John F. Palmer, 1984
[2] Available solution in the Embedded/Control Systems domain is the Scode tool by ETAS: https://www.etas.com/download-

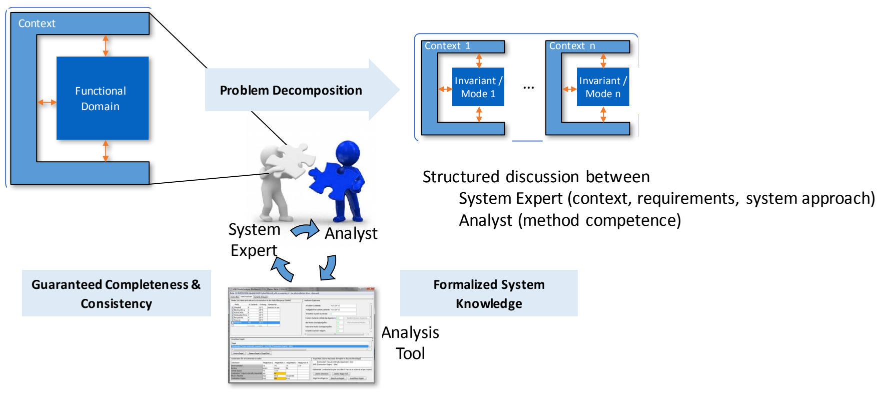

Context
Functional Domain
Problem Decomposition
Context 1
Invariant / Mode 1
...
Context n
Invariant / Mode n
Structured discussion between
System Expert (context, requirements, system approach)
Analyst (method competence)
System Expert
Analyst
Guaranteed Completeness & Consistency
Formalized System Knowledge
Analysis Tool

Figure 6: Essential Analysis Method

2.2.2 QoS management and link between composition and certification

The key factor to correct manage QoS and certification issues for the expert is the ***separation of concerns*** between function and implementation. This aspect is also corroborated by the method suggested for the functional domain (see previous section), which of course can be used only if function and implementation are distinguished in the overall design process. Once again separation of concerns is of primary importance since SW components are often polluted by implicit assumptions that only hold for a specific target platform / middleware. More concretely the separation of concerns principle can be successfully pursued developing implementation-agnostic specifications, i.e. developing applications against **Abstract Interfaces** that are guaranteed on the target platform by **Platform Specific Implementations**.

In the context of QoS management those Abstract Interfaces usually rely on tailored platform mechanism matching specific Models of Computation, i.e. they explicitly refer to an execution model. Another important aspect is that the Abstract Interfaces must expose stable unambiguous semantics including non-functional properties. Obviously, to be useful in practice, the Abstract Interfaces must be verifiable and efficiently implementable.

Example 1. Logical Execution Time

The Logical Execution Time (LET) paradigm (Figure 7) decouples logical timing structure from physical execution resulting in portability, composability and deterministic communication between concurrent functional units. The LET paradigm solves the problem of software distribution on multi-core platforms and represent a strong-base argument for certification[3].

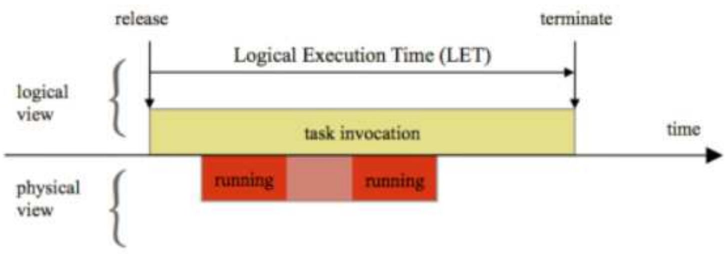


Figure 7: Logical Execution Time

center/files/products_RTA_Software_Products/Whitepaper_SCODE_2016_12_19.pdf

[3] Derler et al.: Simulation of LET Models in Simulink and Ptolemy Monterey. Workshop 2008: Foundations of Computer Software. Future Trends and Techniques for Development pp 83-92.

Example 2. Reservation-based Scheduling

In robotics, most approaches are agnostic to OS mechanisms laying below the middleware layer, so that temporal behaviour is accidental and difficult (or impossible) to predict. As a matter of fact, standard scheduling approaches used in the embedded systems domain (such as Rate Monotonic Scheduling) are not adequate for robotic applications with dynamic resource requirements, leading to **strongly varying or unknown response times**. In the current state of practice, system integration in robotics is often achieved by drastic overprovisioning of computational resources. While such an approach is adequate for research prototypes, product engineering must rely on more systematic approaches leading to provably guaranteed temporal properties (for cost and certification reasons).

Reservation-Based Scheduling (RBS) naturally extends the LET paradigm to the execution management domain. RBS represents a comprehensible abstraction for handling computing resources enabling composability. With RBS, processor capacity is viewed as a quantifiable resource that can be reserved like physical memory. A task receiving a fraction U (<1) of the processor capacity behaves as if it were executing alone on a U times slower processor. Composability is achieved by temporal isolation: an application has access to reservation regardless of the other application executed in the system. Interestingly RBS can be dimensioned for average case (avoiding worst-case design) while improving overall utilization (no idling of cores like with time-triggered approaches).

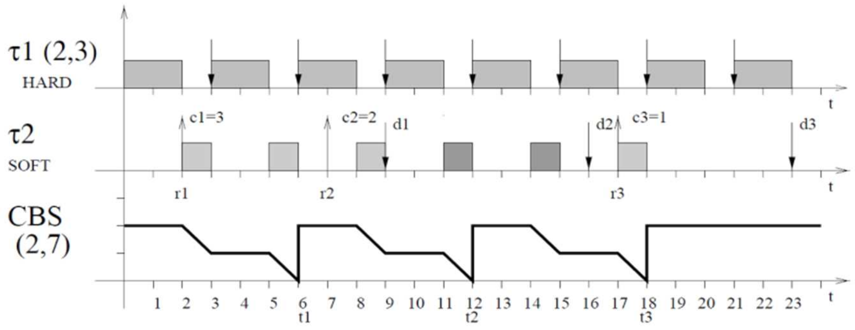


Figure 8: Reservation Based-Scheduling using the example of the Constant Bandwidth Server (CBS)[4]

RBS allows to develop applications independently and integrate them at the end, since reasoning about functional and non-functional properties is possible before integration. This leads to a huge gain in productivity and a string base for certification.

2.3 Jan Broenink : Multi-Paradigm Modelling for Cyber-Physical Systems

Robots can be viewed as a specific class of Cyber-Physical Systems, where the total system (cyber and physical) must be integrally treated and where safety aspects are relevant. The combination among the physical design (mechanical and electrical) and the cyber part (control software) must be handled properly, taking into consideration that after the transformation of signals to data a communication network is in general involved (Networked CPS). To handle such complexity a multi-paradigm modelling is proposed. Different kind of models must be then used to treat the total system. Used models differentiate each other in terms of modelling principles and Models of Computation. In the realm of Discrete Event models for instance different types of models can be found such as State Machine-like models, Process diagrams (block diagrams-like) and hardware description formalisms. Discrete Time models can either consider fixed Time Events or Variable Time Events. Other aspects define the model of computation such as the Communication model i.e. asynchronous (buffer) vs

[4] Luca Abeni, Giorgio Buttazzo : Integrating multimedia applications in hard real-time systems, Real-Time Systems Symposium (RTSS), 1998.

synchronous (rendezvous) and the level of synchronicity between calculations (asynchronous vs synchronous).

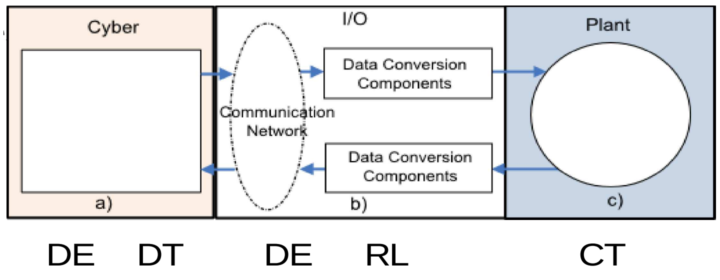


Figure 9: Combined Modelling

Figure 9 shows the different modelling paradigms chosen for the different parts of a cyber-physical system, ranging from Discrete Event (DE) to Continuous Time (CT).

Figure 10 specifies the combination of models used for the different activities involved in the design process. The software architecture (I-a), includes logic for decisions (sequence control) and strategy algorithms (supervisory). To model a software architecture Discrete Event/Time modelling is used, e.g. finite state machines for decisions and software modules (data-flow) for strategy algorithms. Control algorithms (I-b) endows a tighter notion of real-time, so that Discrete Time is used for loop-control algorithms and Continuous Time is used for plant modelling. These models are used in combination for verification and simulation (II) before software implementation synthesis (III).

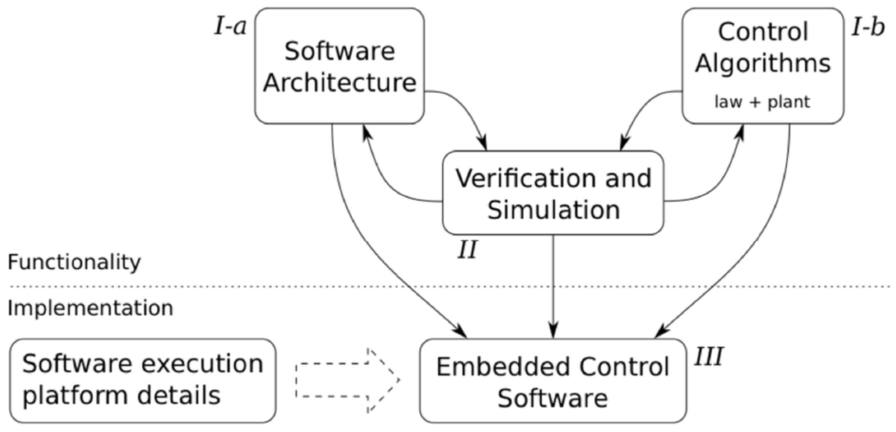


Figure 10: Combined Modelling and Design Process

Jan Broenink presents his specific choice for models, targeting graphical languages:

- Cyber part relies on a graphical representation of Communicating Sequential Processes (gCSP). One of the fundamental features of CSP is that it can serve as a notation for describing concurrent and communicating processes at different levels of abstraction, using different communication models as for instance Rendezvous communication.

- Physical part relies on Bond graphs. A bond graph is a graphical representation of a physical dynamic system. It allows the conversion of the system into a state-space representation.

Both graphical representations are similar to a block diagram or signal-flow graph; with the major difference that the arcs in bond graphs represent bi-directional exchange of physical energy, while those in block diagrams and signal-flow graphs represent directional flow of information.

Models are then support for a concurrent design flow where software, electronics, control and mechanics design run concurrently.

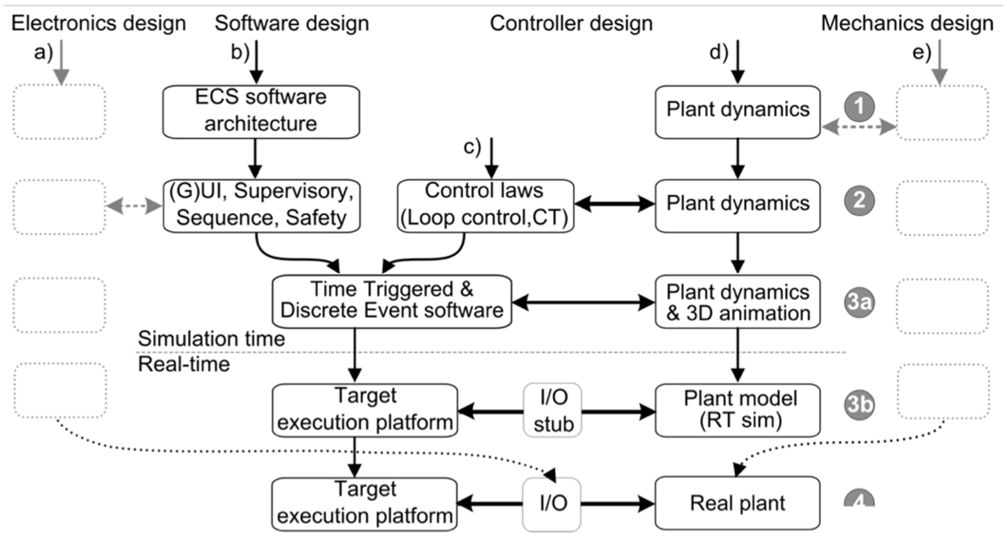

Electronics design
Software design
Controller design
Mechanics design
a)
b)
c)
d)
e)
ECS software architecture
Plant dynamics
1
(G)UI, Supervisory, Sequence, Safety
Control laws (Loop control,CT)
Plant dynamics
2
Time Triggered & Discrete Event software
Plant dynamics & 3D animation
3a
Simulation time
Real-time
Target execution platform
I/O stub
Plant model (RT sim)
3b
Target execution platform
I/O
Real plant
4

Figure 11: Concurrent Design Flow

The models are then refined step-wise

1. Architecture and Dynamic Behavior
2. Model-based control-law design
3. Software, functional co-simulation and real-time specification/simulation of the implementation
4. First-time right realization

While the importance of models is undeniable, it is important to make a distinction between languages vs techniques/methods vs tools to correctly use them during the design flow. Formalisms/Languages are instruments to exploit models: they provide expressiveness to write down models (syntax and semantics). Techniques, pertaining to the realm of model-driven engineering techniques, fall into two main categories: transformation techniques to transform models (expressed in a given formalism) to other models (possibly expressed in a different formalism) and techniques to give insights or retrieve information captured by the models. Methods have a larger scope than techniques, they pertain to reasoning frameworks or design approaches (e.g. the BIP framework presented by Saddek Bensalem). Tools, finally, support methods and implements techniques. Decoupling techniques/methods from tools allows focusing on methods instead of getting lost in tools implementation.

The link between languages and so-called meta-models is of paramount importance. A meta-model is a model of models, i.e. a meta-model defines the language used to write a model. A meta-model indeed specifies rules for checking the correctness of models and represents a basis for tools, specifically for editors and compilers. Common ground between different meta-models can be found in meta-meta-models (a meta-model conforms to a meta-meta-model), while transformations between models are ruled by a transformation among corresponding meta-models.

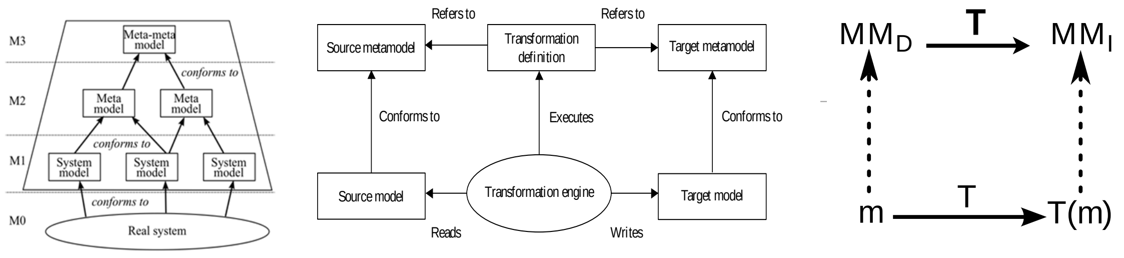


Figure 12: Models Pyramid and Model transformations

2.4 Jurgen Bock: Industry 4.0 as paradigm of digitally connected components.

Industry 4.0 (I4.0) is a term coined in Germany to refer to the fourth industrial revolution. This is understood as the application of concepts such as Internet of Things (IoS), Cyber-physical Systems (CPS), the Internet of Services (IoS) and data-driven architectures in the real industry.

2.4.1 Background: The I4.0 component and the reference model

The Reference Architecture Model for Industry 4.0 (RAMI 4.0 describes fundamental aspects of the Industry 4.0: it illustrates the connection between IT, manufacturers/plants and product life cycle through a three- dimensional space. Each dimension shows a particular part of these worlds divided into different layers as depicted in Figure 13. Left vertical axis represents IT perspective which is comprised of various layers such as business, functional, information, etc. These layers correspond to the IT way of thinking where complex projects are decomposed into smaller manageable parts. In the left hand, horizontal axis is displayed the product life cycle where Type and Instance are distinguished as two main concepts. The model allows the representation of the data gathered during the entire life cycle. Along with the right hand horizontal axis the location of the functionalities and responsibilities are given in the hierarchical organization. The model broadens the hierarchical levels of IEC 62264 1 by adding the Product or a workpiece level at the bottom, and the Connected World goes beyond the boundaries of the individual factory at the top.

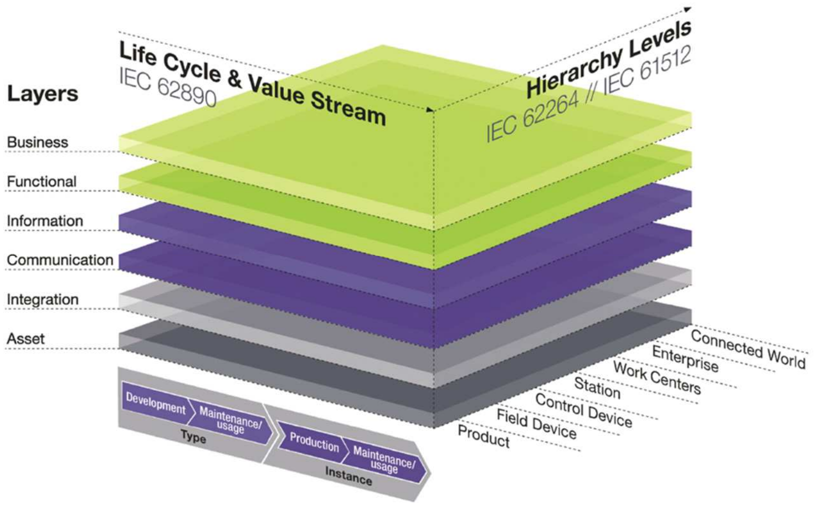


Figure 13: The reference model of Industrie 4.0

A component is a basic concept in Industry 4.0. It is used as a model for representing the properties of real objects in a production environment connected with virtual objects and processes (a CPS system). It is comprised of two foundational elements: one or more assets and Administrative Shell surrounding the assets (Figure 14).

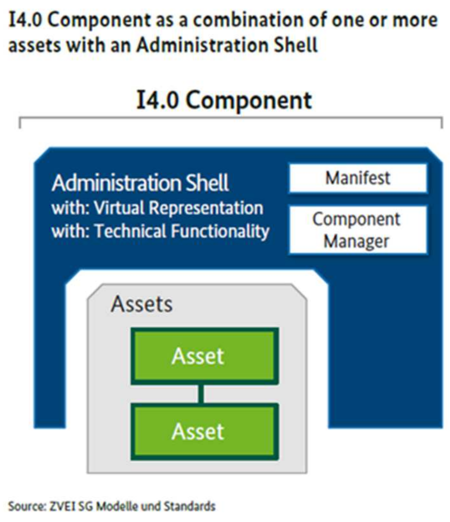

I4.0 Component as a combination of one or more assets with an Administration Shell
I4.0 Component
Administration Shell
with: Virtual Representation
with: Technical Functionality
Manifest
Component Manager
Assets
Asset
Asset
Source: ZVEI SG Modelle und Standards

Figure 14: I4.0 component

An I4.0 component can be a production system, an individual machine, an assembly inside a machine or a software platform. Indeed, I4.0 component can be on different hierarchy levels (product, field device, …, enterprise, connected world). A basic prerequisite is that I4.0 components are able to communicate and understand each other for cooperation scenarios. To this end I4.0 components need to have self-X properties (starting off with self-description) and need to interact. Implementation of interaction is often based on OPC-UA concepts.

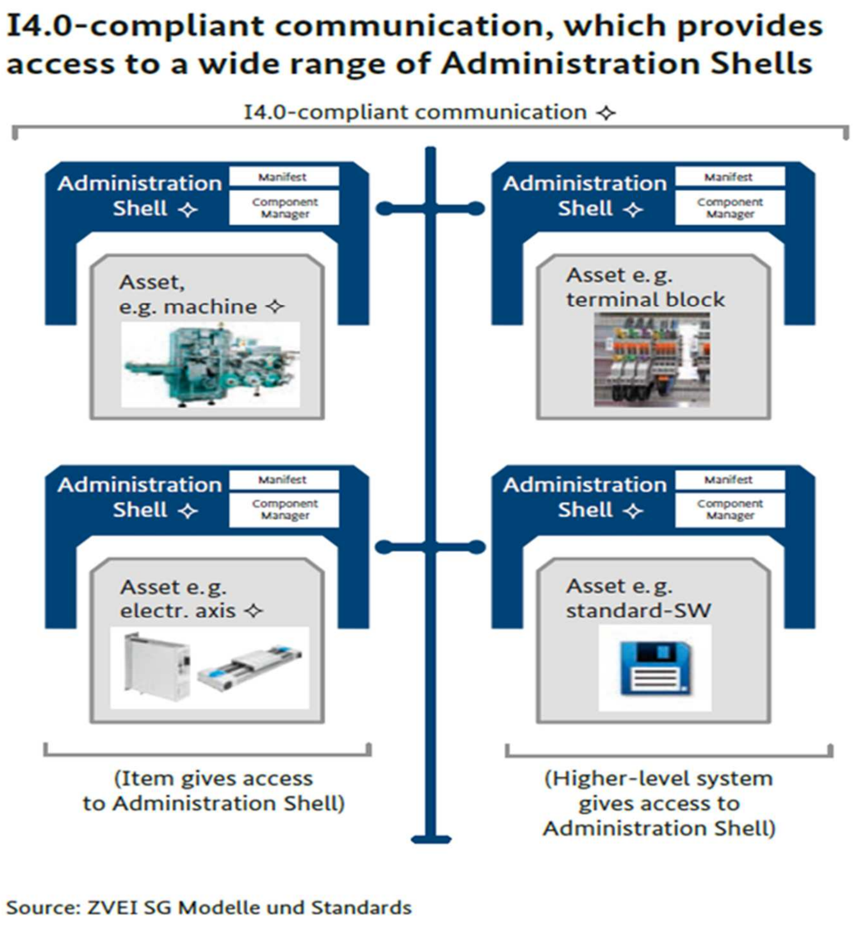

I4.0-compliant communication, which provides access to a wide range of Administration Shells
I4.0-compliant communication
Administration Shell
Manifest
Component Manager
Asset, e.g. machine
Asset e.g. terminal block
Asset e.g. electr. axis
Asset e.g. standard-SW
(Item gives access to Administration Shell)
(Higher-level system gives access to Administration Shell)
Source: ZVEI SG Modelle und Standards

Figure 15: I4.0 components interaction

2.4.2 Interaction model and message-based communication

I4.0 components shall be able to interact with each other following an interaction model Figure 16. Within this model, we can find a set of typical interaction patterns, namely:

- Identification
- Negotiation of security measures
- Request for a task (is a particular task executable?)
- Negotiation of a task
- Order and execution of a task
- Report of errors

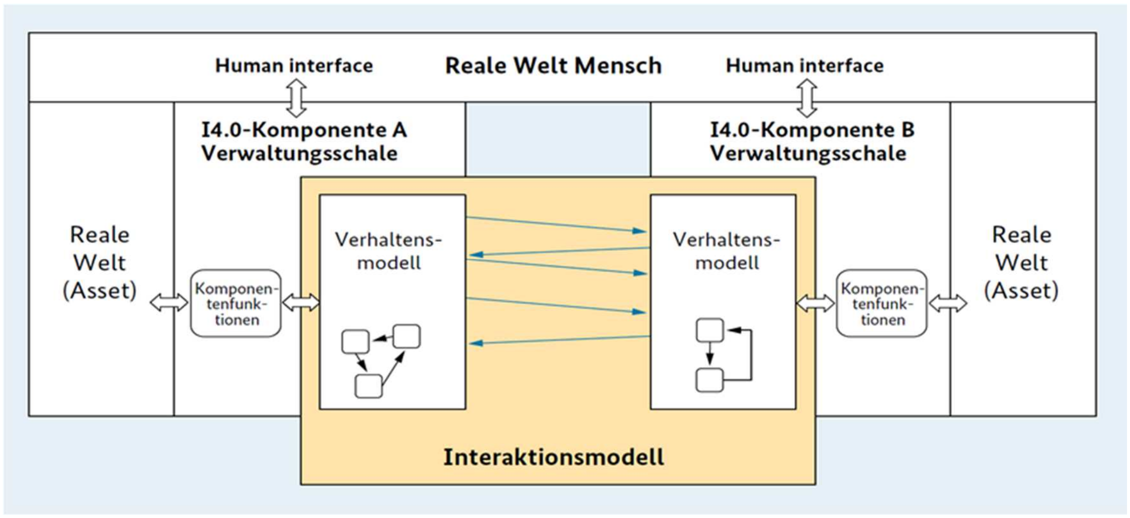


Figure 16: Interaction model[5]

The point-to-point communication is message-based. Both base ontology and domain specific ontologies are considered. Base ontology provides a vocabulary for message format description while the domain ontology provides a vocabulary for message content description.

[5] Bock, J.; Diedrich, C.; Hänisch, R.; Kraft, A.; Neidig, J.; Niggemann, O.; Pethig, F.; Reich, J.; Schulz, T.; Vollmar, F. & Vialkowitsch, J. Weiterentwicklung des Interaktionsmodells für Industrie-4.0-Komponenten Plattform Industrie 4.0, 2016

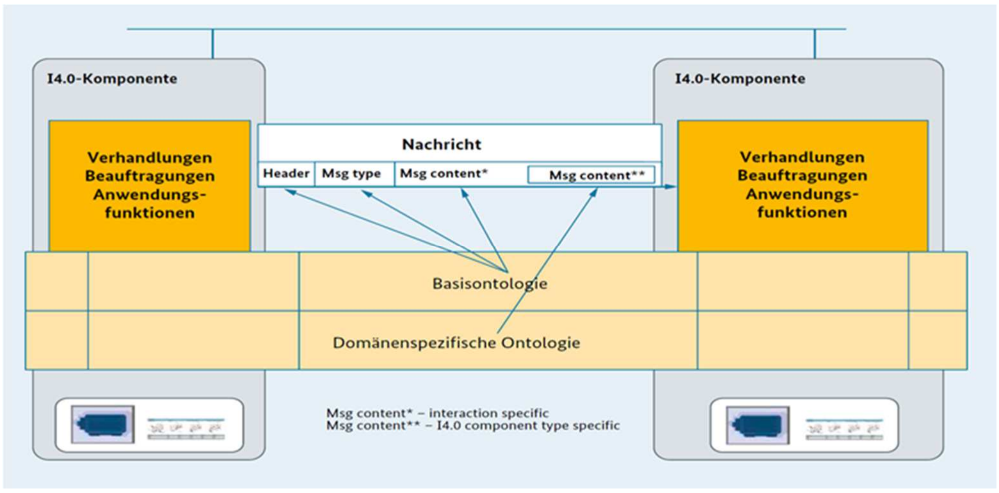


Figure 17: Message-based Communication[5]

2.4.3 OPC-UA

OPC-UA is often the preferred choice to implement communication between I4.0 components. OPC-UA has a client-server architecture, but a publish-subscribe architecture is currently under specification. One of the fundamental concepts in OPC UA is information modelling that can be used to describe the component (provide a structured access to data and methods). The semantics of information models is not formal, but there are companion standards that describe common structures for information models in specific domains or for specific purposes (e.g. Companion Standard "Device Integration" for field devices). OPC UA offers various services, such as security and discovery services.

2.4.4 Composition of Industry 4.0 components

Assets can be arranged freely (Figure 18); different composition patterns might be possible. Composition must obey to the following rules:

- Components to be composed should be Industry 4.0 components
- The Asset Administration Shell should be a standardized interface
- There should be an interaction model for Industry 4.0 components
- Asset Administration Shells can contain sub-models and are thus flexible
- Sub-models can be used for various composition-related aspects, e.g. self-description, negotiation (QoS, tasks, etc.)
- The implementation should be based on OPC UA

The priority to start off with Asset Administration Shells based on OPC UA and subsequently implement sub-models.

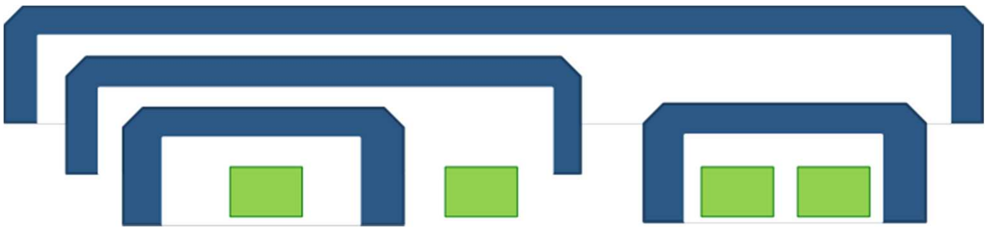


Figure 18: Composition of assets and components

2.4.5 Semantics and Vocabularies

The asset administration shell (AAS) is a generic interface to anything that has a value (asset) for an owner. Every AAS provides a basic set of information about the asset. An added value is only generated if the basic set of data is extended with domain specific data. Therefore, in the header of the AAS a statement is made that the AAS supports a specific model.

In a first step, semantics will be based on shared vocabularies. Current parameters and states are provided by "properties", more precisely, by property value statements. Every property is defined in a dictionary, e.g., ecl@ss or Common Data Dictionary (CDD). In these dictionary, it is actually defined what for example a "pipe diameter" is.

Expressive formal semantics is currently missing but AAS offer a way to incorporate expressive formal semantics: Messages in an interaction model can refer to externally declared properties, e.g. in eCl@ss, or more expressive ontologies. At implementation level this is immediately reflected through OPC UA information models linking to expressive ontologies, OWL, etc.

Currently, different groups are looking for domain specific properties (e.g., drive engineering) in order to insert them into dictionaries like ecl@ss.

2.4.6 QoS management

Let a formal description for QoS be part of the Asset Administration Shell through the definition of a sub-model, then use the interaction model in combination with the sub-model to negotiate QoS properties.

2.4.7 Tooling

While the AAS specification is pretty advanced, tools are not yet available.

3 Current design methodologies assessment

In this section, we report the assessment that each expert provided on current development methodologies. The assessment pertains to the methodologies presented by each expert. For each methodology, the consortium asked for potential risks in applying the methodology, current pain-points, the focus and priorities covered, the suggested good practices, tooling and what in the opinion of the expert are bad practices. Recommendations for the RobMoSys projects are then presented in the last column.

Risks	Pain Points	Priorities	Bad practices/wasted time	Good Practices	Tools	Recommendations
SADDEK						
Verification complexity	State space explosion	- Correct-by-construction Design - Composability rules	Existing development methodologies are of limited interest for systems. They prescribe only general principles and fail to provide rigorous support and guidance. Correctness-by-checking contributes to trustworthiness but it is limited to requirements that can be formalized and checked efficiently (mainly verification of functional properties for application software). For the same reasons, its application to optimization requirements is limited to the validation of scheduling and resource management policies on abstract system models.	- Component/model-based design - Compositional Verification - Assume-Guarantee approach - Incremental Verification - Model checking is applied only to medium size systems when it is possible to make automated proofs or when the cost of faults is high.	- BIP framework - Metropolis Framework	Putting Correctness-by-Construction into Practice: - Horizontal correctness: the construction process of component C is bottom-up. Increasingly complex composite components are built from atomic components by using glue operators. Two principles can be used in this process to obtain a component meeting P: property enforcement and property composability. - Vertical correctness, we need to develop component-refinement tools to allow downstream movement in the abstraction hierarchy from application software modeled in the chosen component framework. - The requirement to deal with a small number of types of components is essential for the formalization of the composition rules between components. Frameworks with a large number of types of components are badly amenable to formalization.

Figure 19: Saddek Bensalem's Assessment

Risks	Pain Points	Priorities	Bad practices/wasted time	Good Practices	Tools	Recommendations
JAN						
Unbalanced solutions due to unawareness of capabilities and restrictions of computing resources	Not appreciating / understanding the value of composition / architecture from a reusability point of view. Application experts and component providers still have difficulty understanding each other.	Vocabulary / ontology; Explain stacks / concerns such that both application experts and component providers can understand each other and benefit from each other; Provide instruments to let roboticists get benefit from the above	Ignore the effect of implementations (i.e. non-idealness of computers / networks). Ignore the viewpoint of the roboticists whose focus is primary on computation	Use declarative models: Bond Graphs, Communicating Sequential Processes in my case Verification and Co-simulation, especially during design. Step-wise refinement as design "guide"	gCSP model checker / co-simulation 20-sim for bond-graph models and control law design. Automatic code Generation from block diagrams Execution lib providing concurrent execution ROS etc, DDS	Separate the language, and methods from the Tools Push way of working as: Model declarative; Refine stepwise; Automatically generate procedural code; Verify / Co-simulate during design...

Figure 20: Jan Broenink's Assessment

Risks	Pain Points	Priorities	Bad practices/wasted time	Good Practices	Tools	Recommendations
ARNE						
Artificial Complexity Demonstrator driven implementation, lack of systematic engineering practices following predefined structures Reuse/portability of SW components not widely recognized as design goals in robotics	SW components implemented with implicit assumptions which are only true on specific platforms (e.g. ROS) → SW components not portable among target platforms No separation of concerns: SW component implementations are overly complex because they cannot assume any guarantees (e.g. QoS) from the underlying platform Missing link (or mutual ignorance) between function and performance	Consistent vocabulary of the different domains, the structuring of the motion stack serves as guiding example Right Abstractions e.g. Abstract Machine Interfaces to handle QoS attributes on application level. Thereby most importantly, abstractions for handling computational / communication resources Definition of the computational model on system level (i.e. how and when do components exchange data, when are computations executed, etc.) Definition of end-to-end timing constraints for cause-effect-chains spanning multiple SW components Configuration and variability of SW components (context: using a generic component in the context of different robots) Safety handling	Models with non-adequate abstraction level: e.g. Code Generation from UML models (too concrete), Boxes and Lines (too unconcrete) Non-constructive approaches (too many iterations) Approaches by-passing architectural patterns (e.g. complex drivers in AUTOSAR)	Morphological method to structure a functional domain Logical Execution Time for composability and portability on communication level (good 4 certification) Reservation Based Scheduling Schemes for composability on computational level (good 4 certification) Methodology driven structured engineering approaches such as AUTOSAR in automotive (freedom from choice) Declarative models including methods/tools to derive procedural realizations GALS structure on system level. Ignore "minor" coupling effects on functional level to derive flexible implementation with sufficient degrees of freedom: "approximate refinement".	Essential Analysis Tool LITMUS^RT as backbone for execution container Tool for specifying cause-effect-chains: as link between the functional domain and the execution domain Real-time analysis tools like SymTA/S, Inchron	Ask to extension to existing implementations e.g. LINUX extension for RBS Developed concept should be technology agnostic but show cased using concrete technology like ROS, ROS2.0 Different aspects in the call should be illustrated with concrete problems (cf. end-2-end latency with CFS vs RBS) Prefer declarative approaches over procedural approaches, since the former better supports composability Search for executable models (i.e. defining clear computational models) on component level. Ask explicitly for quantitative evaluation of the proposed method/tool/approach compared to the current state of practice (i.e. show the benefit)

Risks	Pain Points	Priorities	Bad practices/wasted time	Good Practices	Tools	Recommendations
ARNE						
Artificial Complexity Demonstrator driven implementation, lack of systematic engineering practices following predefined structures Reuse/portability of SW components not widely recognized as design goals in robotics	SW components implemented with implicit assumptions which are only true on specific platforms (e.g. ROS) → SW components not portable among target platforms No separation of concerns: SW component implementations are overly complex because they cannot assume any guarantees (e.g. QoS) from the underlying platform Missing link (or mutual ignorance) between function and performance	Consistent vocabulary of the different domains, the structuring of the motion stack serves as guiding example Right Abstractions e.g. Abstract Machine Interfaces to handle QoS attributes on application level. Thereby most importantly, abstractions for handling computational / communication resources Definition of the computational model on system level (i.e. how and when do components exchange data, when are computations executed, etc.) Definition of end-to-end timing constraints for cause-effect-chains spanning multiple SW components Configuration and variability of SW components (context: using a generic component in the context of different robots) Safety handling	Models with non-adequate abstraction level: e.g. Code Generation from UML models (too concrete), Boxes and Lines (too unconcrete) Non-constructive approaches (too many iterations) Approaches by-passing architectural patterns (e.g. complex drivers in AUTOSAR)	Morphological method to structure a functional domain Logical Execution Time for composability and portability on communication level (good 4 certification) Reservation Based Scheduling Schemes for composability on computational level (good 4 certification) Methodology driven structured engineering approaches such as AUTOSAR in automotive (freedom from choice) Declarative models including methods/tools to derive procedural realizations GALS structure on system level. Ignore "minor" coupling effects on functional level to derive flexible implementation with sufficient degrees of freedom: "approximate refinement".	Essential Analysis Tool LITMUS^RT as backbone for execution container Tool for specifying cause-effect-chains: as link between the functional domain and the execution domain Real-time analysis tools like SymTA/S, Inchron	Ask to extension to existing implementations e.g. LINUX extension for RBS Developed concept should be technology agnostic but show cased using concrete technology like ROS, ROS2.0 Different aspects in the call should be illustrated with concrete problems (cf. end-2-end latency with CFS vs RBS) Prefer declarative approaches over procedural approaches, since the former better supports composability Search for executable models (i.e. defining clear computational models) on component level. Ask explicitly for quantitative evaluation of the proposed method/tool/approach compared to the current state of practice (i.e. show the benefit)

Figure 21: Arne Hamann's Assessment

Risks	Pain Points	Priorities	Not Relevant	Bad practices/wasted time	Good Practices	Tools	Recommendations
JURGEN							
Being too abstract for system integrators, end users, component providers, …, to transfer into real systems	Currently it's a difficult ongoing discussion, on where and how the asset administration shell is being implemented. There should be a distinction between the role, type, instance, and real physical asset of any I4.0 component. Strictly speaking, the role "robot with properties 123, type "KUKA KR6", instance "KUKA KR6 with S/N 987", and the real physical robot, are three assets, but do they all need administration shells? (These are ongoing discussions in I4.0 consortia and projects.)	Detailed specification and implementation of the Asset Administration Shell Base Ontology for interactions (Messages) Abstract communication primitives Semantic interoperability Coordination (interaction models specified as state machines) Safety and security	Real-time (in many higher level coordination tasks) Lower levels (Hardware, OS, Exec. Cont.) might not have to be seen as I4.0 components, as they are managed by the asset providers	Trying to be too disruptive on the shopfloor (legacy systems) will not be accepted	Quickly get to the point of having an implementation (with quantifiable analysis) Agreeing on a particular standard toolchain is essential (e.g. OPC UA in Industry 40)	OPC UA (SDKs and software stacks), tools for authoring OPC UA information models (e.g. UaModeller) Tools for creating / configuring I4.0 concepts, e.g. AAS, Interaction Manager, etc.	Focus on small and concrete use cases (concrete use case requested!)

Figure 22: Jurgen Bock's Assessment

4 Synthesis on recommendations for RobMoSys

In this section, we provide a synthesis of the Experts recommendations relevant for the RobMoSys Project.

A first aspect pertains to the nature of the models employed. From experts' recommendations, we can derive the following conclusions:

- Prefer declarative models (including methods/tools to derive procedural realizations) over procedural models since declarative models better support composability;
- Definition of a model of computation (MoC) on system level, i.e. how and when do components exchange data, when are computations executed, etc. The use of a MoC enables verification and co-simulation during design.
- Enable refinements to allow downstream movement in the abstraction hierarchy. GALS (Globally Asynchronous and Locally Synchronous) is a paradigm on system-level allowing refinements with a certain degree of freedom towards implementations.
- Small number of types of components in order to manage the formalization of composition/composability rules and properties.

A second aspect is related to good practices to employ during the development process:

- Put correctness by construction into practice supporting both bottom-up construction (component built out of atomic components) and top-down approaches (refinements) via composition rules and step-wise correct refinement of components

- Push verification and co-simulation activities during design; use model-checking only on small models, composable/incremental verification
- Push the concept of concurrent design (e.g. mechanical, electronical, software)
- Handle safety and security aspects as soon as possible and not as an afterthought

A third aspect pertains to tooling

- Separate the language from the methods and from the tool
- Agree on a standard tool-chain and a common vocabulary for interoperability: at least syntax, better to have semantic interoperability
- Make use of tooling to formalize and assess the structuring of domain knowledge: low dependency/overlapping between concepts

Final remarks have been also provided for use-cases

- Focus on small and concrete use cases
- Developed concepts should be technology agnostic but show-cased using concrete technology (e.g. real OS, middleware, etc.)
- Provide assessments to show the benefit of the concept/method/tool with respect to current state of the art