Honda Research Institute EU
Honda Research Institute Europe GmbH
https://www.honda-ri.de/

A Language for Formal Design of Embedded Intelligence Research Systems

Benjamin Dittes, Christian Goerick

2011

Preprint:

This is an accepted article published in Robotics and Autonomous Systems. The final authenticated version is available online at: https://doi.org/[DOI not available]

A Language for Formal Design of Embedded Intelligence Research Systems

Benjamin Dittes[a,*], Christian Goerick[a]

[a] *Honda Research Institute Europe GmbH, Carl-Legien-Str. 30, 67073 Offenbach, Germany.*

Abstract

The construction of complex artifacts of artificial intelligence requires large-scale system integration and collaboration. System architectures are a central issue to enable this process. To develop these, hypotheses must be formulated, validated and evolved. We therefore present SYSTEMATICA 2D, a formalism suitable for both flexible description of hierarchical architecture concepts as well as functional design of the resulting system integration process. We motivate the approach and relate it to other formal descriptions by means of a new formalization measure. It consists of a set of criteria to evaluate how well a formalism supports the expression, construction and reuse of intelligent systems. SYSTEMATICA 2D is compared with existing formalization languages under this measure and shown to have at least their level of expression. In addition, the system properties of incremental composition, partial testability and global deadlock-free operation are formally defined and proven in the formalism.

Keywords: System Integration, Hierarchical Architecture, Formal Description, System Design

1. Introduction

In the strive to create complex artifacts with artificial intelligence (which together we will call "embedded intelligence" or EI) an interplay of a great number of disciplines is required. Although the actual organization of subsystems in any implemented system may vary, these can be roughly categorized into designing or learning of functionalities for

I information acquisition from the world,

II internal information processing,

III generating actions in the world,

IV integration of subsystems to create a desired overall system behavior.

Popular domains include robotics, driver assistance or autonomous driving as well as virtual agents. In addition to countless efforts in specific areas of I, II and III, cognitive architectures as a means to integrate several subsystems to an intelligent artifact (EI) are receiving steadily increasing attention, see [1] for a survey. However, these system integration processes suffer from three handicaps:

- A multitude of notations, making them difficult to relate to one another,
- a decomposition which is typically not tailored to the needs of interacting scientists and, arising from that,
- a resulting system which makes reuse of larger building blocks difficult.

In this contribution we will address these issues by two means: First, a detailed analysis of the specific challenges and requirements of EI system design and construction will lead to a set of criteria for evaluating the suitability of formal notations for EI systems. The application of these criteria to a set of existing formalisms will yield specific shortcomings to be overcome and will serve as a basis for comparison. Second, we will present and discuss our proposal for a new system design formalism, called SYSTEMATICA 2D, suitable for both system description and comparison as well as for system design and construction.

Since the comparison of system formalisms is a central part of this contribution, the related work presented in section 2 will only cover the wider area of EI system notations and taxonomies for such notations. A detailed analysis of the relation of this contribution to specific existing approaches from software engineering, software infrastructure and functional modeling will follow in later sections. The evaluation criteria are introduced in section 3, following a discussion of the relation of EI system integration to software engineering practices (e. g. UML[2]) and to software infrastructures (e. g. CAST[3], XCF[4], YARP[5], etc.). In section 4 several existing formalisms are then evaluated along those criteria: the commonly used ad-hoc 'boxes and arrows', 3-Tier[6], CogAff[7] and SYSTEMATICA[8]. A conclusion at the end of section 4

*Corresponding author

Email addresses: benjamin.dittes@honda-ri.de (Benjamin Dittes), christian.goerick@honda-ri.de (Christian Goerick)

summarizes the lessons learnt from defining and applying the measure criteria to existing formalisms.

In the second half of this contribution, we start by introducing the proposed new formalism, SYSTEMATICA 2D, in section 5. Central elements of the language are units, relating to individual scientist's contributions to the whole system, and their dependencies and communication patterns; based on the formal description of these elements, a set of system properties can be defined and proven, most notably incremental construction and global deadlock-free operation. Section 6 then focuses on comparing the new formalism to the evaluated existing ones in two ways: first by 'translating' designs from 3-Tier, CogAff and SYSTEMATICA to SYSTEMATICA 2D and second by applying the defined measure criteria to our new formalism. Finally, the discussion in section 7 will cover the mapping from SYS2D design to software infrastructure, the specific structural bias imposed by SYSTEMATICA 2D on the example of several standard design patterns and the descriptive abilities of SYSTEMATICA 2D in terms of distinguishing integration approaches.

2. Related Work

Two areas of related work are relevant to this contribution: taxonomies or measures of formal notations and specific notations for intelligent artifacts.

Attempts at formulating measures, classification frameworks or taxonomies of formal notations are sparse, qualitative and, for the most part, too generic to be of value when judging the benefit of a formalism for EI systems. One prominent example is the work of Medvidovic et al.[9] giving a qualitative comparison of software architecture description languages (ADLs). In this work, they provide a fixed set of description elements which an ADL needs to have: Components, Connectors and Architecture Configurations. Based on a detailed discussion of each of these three elements, the authors evaluate several languages used to describe software architectures, e. g. Rapide[10], C2, LILEANNA and ACME[11], and determine which of them can be considered an ADL. One of the conclusions we can strongly agree on is that the defining property of an ADL is the ability to describe full system configurations (i. e. the pattern in which components and connectors are combined to form a system), in addition to the system elements alone. However, although the work claims to allow comparing ADLs, it does not provide a measure to judge which kinds of constraints on the description of a system are suitable for which domain.

For intelligent systems, we see two comparisons of integration approaches and formalisms: Vernon et al.[1] give a survey of recent development in cognitive architectures by analyzing a wide range of approaches and sorting them into three 'paradigms' of cognition (Cognitivist, Emergent and Hybrid). A different approach is pursued by Goerick et al.[8], where a new framework for modeling hierarchical architectures ('SYSTEMATICA', an evolution of the subsumption architecture[12]) is used to express existing cognitive architectures in the same language and compare them on this basis. Both are able to compare existing architectures to one another but they do not evaluate how the elements of the specific description languages affect their cognitive qualities.

Formal notations for intelligent systems today come from three areas. First, there are mathematical formalizations of system component interaction[13, 14, 15]. These allow a formal analysis and proof of interaction properties of components, but there is no evidence that the attached description languages are able to express established cognitive architectures such as 3-Tier[6] or CogAff[16]. Second, architecture description languages are a popular tool in the software engineering domain to describe large software systems, for instance Rapide[10] and ACME[11], but probably the most generic being xADL[17]. These languages contain all relevant elements for describing an architecture but since, to the best of our knowledge, no application of such an ADL to the EI domain has been attempted, it is unclear what bias they can provide for guiding an EI system design in a favorable direction (see discussion of structural bias in section 3.2). It is reasonable to assume that the formalism introduced in our contribution can be expressed using, for instance, xADL, but this is beyond the scope of this work.

Finally, there are specific notations used in intelligent systems[12, 7, 18, 19, 20, 21, 22] or reviews[6] and plans[23] of such. Among these notations, we see two groups: on the one hand there are systems described in a formalism used only once in the paper describing the system, usually (but not always) closely related to the software infrastructure on which the implementation is based —we will refer to these notations collectively as 'Boxes and Arrows'. On the other hand there are systems described in independently introduced notations, we will focus our comparison on 3-Tier[6], used in [20], CogAff[16], used in [7], and SYSTEMATICA[8], used in [19]. We will perform a more detailed analysis of these four notations, 'Boxes and Arrows', 3-Tier, CogAff and SYSTEMATICA—the focus being the comparison of the notations, not of the systems built with them —when we evaluate them in section 4.

To conclude, we can say that there is a great variety in the notations used to describe systems and a very small number of attempts to measure and compare these. In the following, we will therefore start by introducing such a measure while discussing the relation of EI systems, software engineering and software infrastructures. This measure will then be used to evaluate a set of notations and compare them to the new formalism introduced in this contribution.

3. A Measure for System Design Formalisms

As a first step to improve formalization of EI systems we will argue for a set of criteria such design languages should fulfill. In order to arrive at those criteria, the

first question is: What is important for intelligent system integration? We believe the central element is what we will call the 'Hypothesis Test Cycle'. There is, as yet, no clear and established structure or system architecture concept to build intelligent systems. The natural scientific approach to arrive at such a structure is the creation, validation and evolution of hypotheses about it, which together make up this cycle.

Thus, the main motivation behind the search for good formalisms, and therefore the search for criteria to measure them, is to speed up this hypothesis test cycle behind every integration process. Three considerations then shaped these criteria specifically: the difference between intelligent system integration and software engineering, the importance of structural bias and the relation between a formal system design and the software infrastructure used for implementation.

3.1. Why EI System Integration Is Not Software Engineering

What makes EI system integration special? It is, to a large part, software development —and yet to arrive at criteria to judge formalisms specifically for scientific work on research systems we must understand their specific requirements. The basic questions of analysis, design, implementation, deployment and life cycle management of large software systems are not new but answering them in a scientific context, especially w.r.t. large-scale system integration is rarely attempted. Even the hypothesis test cycle resembles, whether intentionally or not, typical software development life cycle methodologies[24] like the spiral model or extreme programming —but in our experience, more often than not the actual choice of method is among the agile models.

We believe there are three fundamentally different constraints which apply to scientific software integration as opposed to industrial software engineering. First, the components of the system to be integrated are rarely finalized when integration starts —neither their theoretical basis nor their implementation. Second, with every scientist being the expert in his or her specific area, and thus for his or her specific part of the system, it is an impossible task for a system designer to plan the integrated system, composed of many state-of-the-art components from many experts, down to the last class or member variable —a level of flexibility which only the specific experts can fill is inevitable. Finally, the process of integration is not separate from each scientist's work on his contribution to the system but intertwined both ways: lessons learnt developing components can influence system design and lessons learnt from running components in the full system can provide new constraints for component development.

These differences lead us to the conclusion that the perfect formalism cannot be the most exact one. Languages like UML[2] are suitable for analysis and design of large software systems, built by dedicated developers based on established principles. Research system integration, on the other hand, requires a focus on expressing system hypothesis and component interconnectivity, but at a level that leaves the necessary room for scientific work (similar arguments are presented in [25]).

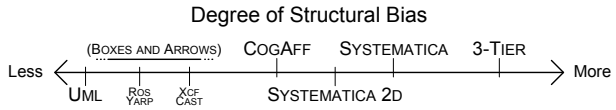

Figure 1: Rough ordering of formalization approaches according to the amount of structural bias.

3.2. The Importance of Structural Bias

If any formalism must allow room for the individual scientist's work, a valid question to ask is what the point of specific formalisms for EI system integration is altogether. In fact, the description of systems and system hypotheses as arbitrary graphs (an approach we will evaluate under the name 'Boxes and Arrows' later) is widely used exactly because it does not limit the range of expression —thus allowing the preferred level of specificity.

However, in giving up the guidance of a specific formalism, all other benefits such a formalism might provide vanish with it. This is mainly important during the implementation phase of a system, but also during design and refactoring a formalism can help to consider details which will be important later.

We call this influence of the formalism on the system integration process 'Structural Bias'. Fig. 1 shows a sorting of the formalisms analyzed more closely in this work according to their degree of structural bias. UML is clearly the most detailed, but also the formalism with the least constraints on the way concepts must be expressed. 'Boxes and Arrows' are similar, with the degree of bias depending on the specific application. Three related formalisms, CogAff, Systematica and 3-Tier will be analyzed in sections 4 according to the criteria defined in the following; the new formalism Systematica 2D introduced in this contribution will be evaluated in section 6.2.

3.3. Relation of System Design to Software Infrastructure

With the declared aim of providing support for the full hypothesis test cycle —including implementation and reuse —the importance of a software infrastructure to run the designed systems cannot be ignored. The question then is how well established infrastructures are suited for system design already, or at least how the design of a system and the infrastructure chosen for its implementation are related.

To this end we will review the types of system targeted, the level of description and the structural bias imposed by six such established infrastructures:

CAST. The CoSy Architecture Schema Toolkit aims at "construction and exploration of information-processing architectures for intelligent systems"[3]. Central mechanism of description is a decomposition of the whole system into sub-architectures running in separate processes, each with a working memory and a set of managed and unmanaged components. Interaction between sub-architectures is done by reading each other's working memories and through a central goal manager.

Beyond the decision about sub-architecture granularity and separation of components into managed and unmanaged, the structural bias is quite low; to understand communication patterns, component interdependencies or the relation between representations a separate system design language would be beneficial.

XCF. The XML enabled Communication Framework[18, 4] aims at providing a simple and standardized approach to distributed processing and memory structures of cognitive systems. Central mechanisms of description are the separation of processing into asynchronously running components, an Active Memory XML server and the communication between those entities using standardized XML messages. Interaction between processing components is done through the active memory by queries and subscriptions to events, potentially coordinated by a central Active Memory Petri-Net Engine.

Except for this very last point (the petri-net) there is a strong similarity of decomposition and description (if not of implementation specifics) between XCF and CAST: within a CAST sub-architecture, the communication pattern of components and working memory is comparable to that between XCF components and the active memory. Above that CAST adds distribution into multiple sub-architectures and a management of goals and XCF adds a more elaborate internal dynamic of the active memory up to a central coordination using petri-nets. However, the structural bias imposed by XCF is not stronger than that of CAST and also here communication patterns and interdependencies (between components) would benefit from a separate design language.

Middleware. A group of very software-oriented infrastructures is spanned by ThinkingCap II[26], YARP[5], ROS[27] and OpenRTM[28]. They all share the basic decomposition of a system into concurrently running processing components and support their configuration, communication and monitoring. All except ThinkingCap support the addition and removal of components at run-time, main differences are in the chosen programming languages and the specific communication protocol (direct or by subscription) and communication method (XML-RPC or custom).

What all of them have in common is that structural bias introduced by the infrastructure is (intentionally) very low. For instance, while an architecture description file (ADF) in ThinkingCap II at least contains a description of the entire system to be run, ROS is only able to determine the specific communication pattern between components by run-time analysis. We therefore conclude that especially for this set of infrastructures, an additional design language determining the relation of components and their communication and dependencies is essential.

Conclusion. It is not our intention to deny that a software infrastructure is essential for moving from a system design to an implemented system. However, the evaluation of structural bias imposed by the analyzed infrastructures has shown that this is not the level where a discussion about design and formal design languages is adequate. It is the goal of most infrastructures to provide the tools for implementing a very large spectrum of possible applications. Guiding this process by enforcing consideration of component dependencies and subsystem separation during the design process (as the measure criteria in the following will make explicit) is the task of a formal design language.

One criterion for such a language (but one of many) must be that it can be mapped to (at least) one software infrastructure, e. g. the example in [26] for ThinkingCap II is based on a 3-Tier design. We will therefore consider the relation between the evaluated formalisms and software infrastructures (where published) when applying the measure in this section. In addition, section 7.1 will discuss the mapping of the SYSTEMATICA 2D formal design language introduced in this paper to a set of software infrastructures.

3.4. Criteria for System Integration Formalisms

Few attempts at establishing a formalism measure have been made (e. g. [9, 1]) for an obvious reason: it will be qualitative and argumentative at best, asking the right questions about a formalism, but without the means to judge their fulfillment quantitatively. We will therefore use it with caution and revisit the lessons we can learn from judging a formalism with this measure when we do so in section 4.

That being said, we will now formulate criteria relating to the two main considerations: criteria A1-A4 specify the required minimum power of expression for the system hypothesis; criteria B1-B5 specify additional structural bias for an efficient hypothesis test cycle, especially regarding implementation.

3.4.1. Flexible Description

> **A1: A formalism should not limit the range of architecture hypotheses that can be expressed** —the structural bias should direct the way *how* hypotheses are expressed (see following criteria), but it should not limit *which* hypotheses are possible.

3.4.2. Meaningful Description

A2: A formalism should not hide information necessary for understanding an architecture hypothesis but try to find an appropriate level of granularity —to support the full hypothesis test cycle, a design must be simple enough to transfer the main idea on the one hand side and detailed enough to aid construction of the system on the other hand side.

3.4.3. Standardized Description

A3: A formalism should use a standardized, unambiguous and intuitive notation to ease discussion and publication —important both for publication of the architecture and for communication with the scientists working on the system.

3.4.4. Description of Interfaces

A4: A formalism should allow specification of the interfaces of system elements (units) whenever they affect at least two collaborating scientists —following the arguments in section 3.1, the main purpose of the design can only be to describe what is between individual scientist's fields of work, most notably their interfaces.

3.4.5. Decomposition to Individuals

B1: A formalism should allow decomposition of the architecture to units for individual scientists —when it comes to implementation, a good formalism will allow individual scientists to work on their units individually until they reach a state that can be integrated; a granularity which is too rough will endanger this separation (this relates to granularity, see A2).

3.4.6. Description of Dependencies

B2: A formalism should allow specifying the dependencies between collaborating scientists and identify strong or loosely coupled interaction —along the same lines as B1, each individual working on a unit should be at least aware of the units required for his or her work.

3.4.7. Translation to Infrastructure

B3: The decomposed units of a formalism should translate into decomposed units of the software infrastructure chosen for implementation —important for both implementation and reuse: during implementation, the translation allows a quick understanding and navigation of the system, during reuse it is always easier to salvage self-contained units than to split them.

3.4.8. Exploitation of Infrastructure

B4: The dependencies and interfaces specified by the formalism should be compatible with the chosen software infrastructure to allow partial testing and graceful degradation, if available —if a design can decompose to individuals (B1) and express their dependencies (B2), it is a direct extension to ask for partial execution of subsets of units in order to allow scientists to test their work in a reduced system or to allow the system to stay functional when some units fail.

3.4.9. Subsystem Separation

B5: A formalism should allow separating an existing system into subsystems and reusing or extending its subsystems by means of the decomposition in the implementation (B3) and the formal description of dependencies (B2) —reuse of single units is good, exploiting decomposition and dependencies to allow reuse of larger building blocks is better.

4. Evaluation of Existing Formalisms

We will now proceed to validate the formulated formalism measure, as well as discuss the range and merit of its application, on the example of four existing techniques for formalizing EI systems: 'Boxes and Arrows' (as an example for the typical ad-hoc approach, used in countless publications), 3-Tier (as a popular example of technical embedded system modeling, see [6, 20]), CogAff (as an example of a biologically motivated design, see [7]) and SYSTEMATICA (as a formal language for analysis of hierarchical architectures, see [8]). We chose these candidates because we believe they cover a wide range of techniques of how systems are *described*; we will focus only on this quality of description, not on the specific systems built with them.

As mentioned in section 3.4, the application of each of the defined criteria is qualitative. To allow comparability, we will apply them based on the following questions:

- Does the formalism explicitly require the information to satisfy the criterion?
- Does the formalism ask relevant questions towards satisfying the criterion?
- Does the formalism imply a bias for system architectures towards or against the criterion?

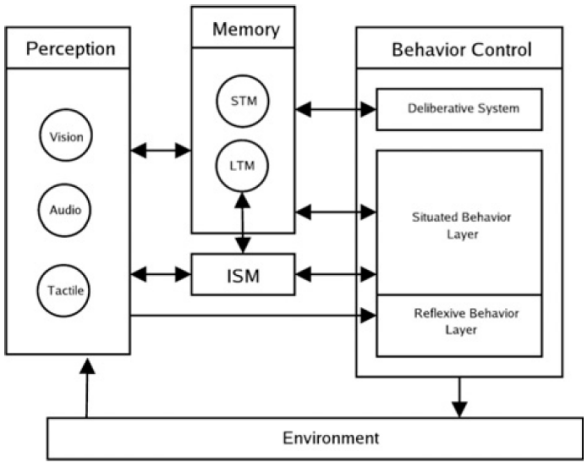

Figure 2: The EGO architecture as an example for the "Boxes and Arrows" formalization approach. Taken from [29].

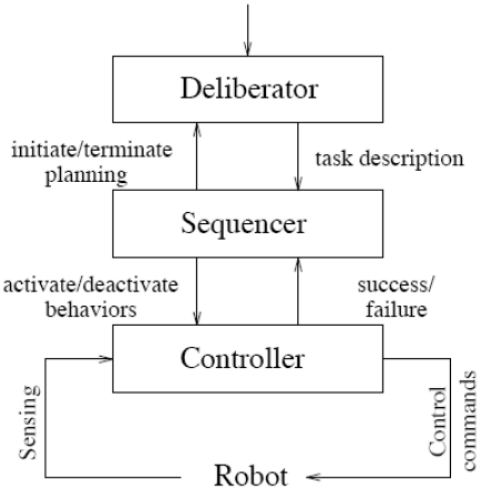

Figure 3: Schematic view of the 3-Tier architecture skeleton as proposed by [6].

4.1. The "Boxes and Arrows" Formalism

A very common approach to system design is the ad-hoc style of drawing boxes and arrows on paper, white board or any other structure-free medium. The first, and valid, question about any more structured system formalism is therefore: "Why is it better than arbitrary boxes and arrows?" Using the formalism criteria introduced in section 3.4 we can formulate the strong and weak points of this kind of ad-hoc formalization (see Fig. 2 for an example).

Our evaluation of the **Boxes and Arrows** formalism thus looks as follows:

A1 (+) The range of expression with arbitrary boxes and arrows is limitless; this is the main advantage of this approach.

A2 (+) The level of description may vary, but the flexibility of the approach allows description at an appropriate level of detail.

A3 (?) Together with a precise description of the meaning of the used shapes, the hypothesis can be discussed or published. Additional effort may also ensure intuitive presentation and remove ambiguities, but the formalism does not provide tools to ensure this.

A4,B (–) Generally speaking, the formalism does not enforce specifying details like dependencies, interfaces or granularity —it does not even force the designer to consider them. Due to the flexibility, a predefined way to translate design units to the infrastructure cannot be ensured.

All these assessments are done for the general case of boxes. Although they are each argumentative, all together give a view of the merits (high flexibility) and drawbacks (limited focus on construction) of the given formalism. Naturally, most other, more detailed formalisms also use specific boxes as the central means of expression —but, by characterizing them more closely, ensure consideration of more criteria.

4.2. 3-Tier Architectures

The class of 3-Tier architectures, as presented by Gat[6] and used more recently e. g. in [20], is mainly used for robot control where reactive and deliberative systems work together (see Fig. 3).

Our evaluation of the **3-Tier** formalism thus looks as follows:

A1,2 (–) The decomposition is fixed to the three main layers for controller, sequencer and deliberator; a different composition or finer description is not intended.

A3 (+) Since the original publication, the 3-Tier approach has been used and the description can therefore be seen as standardized.

A4 (?) The language does not directly formalize the data transmitted between layers, but the nature of all communications is implied by the concept.

B1 (–) The rough, three-part decomposition cannot express the separation of individual scientist's work packages.

B2 (+) Based on the original concept, a tight coupling from bottom to top and a loose coupling from top to bottom are implied.

B3,4 (+) 3-Tier is traditionally used for robotic applications, therefore there are many examples of implementations.

B5 (?) Although the design units can be translated to infrastructure units, their rough granularity makes

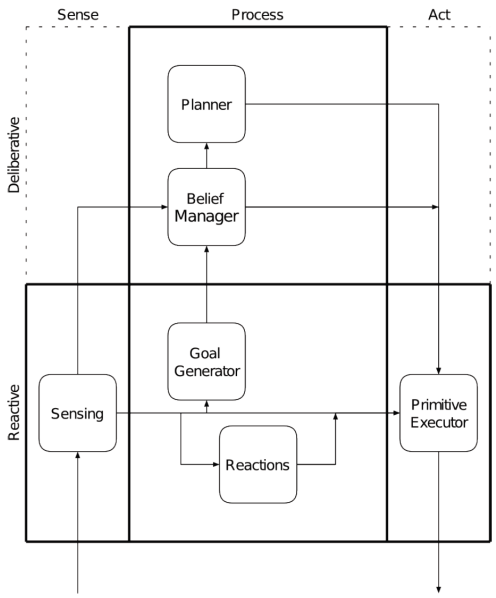

Figure 4: Example of an agent design in the CogAff framework[7].

it unlikely that they can be reused without modification in a subsequent implementation.

4.3. CogAff

The CogAff architecture schema presented by Sloman et al.[16] is a framework for embedding and relating integrated functionalities. Since the schema itself is not mainly a means of specifying systems we look at an application based on CogAff[7] to evaluate the power of this formalism, see Fig. 4.

Our evaluation of the **CogAff** formalism thus looks as follows:

A1,2 (+) The two-dimensional arrangement of units along the axes Sense-Process-Act (horizontal) and Reactive-Deliberative-Meta (vertical) allows a flexible arrangement of integrated functionalities with arbitrary detail. The formalism is restricted to the CogAff domain, but this is not a major restriction for intelligent artifacts.

A3 (?) Apart from their positioning, the description of units and connections is not precisely specified, neither in the CogAff proposal[16] nor in the sample application[7].

A4 (–) A specification of interfaces is not included.

B1,2 (+) Fine decomposition, focusing on single scientists is possible, dependencies are not specified but can be implied from the positioning.

B3,4 (+) Although no specific infrastructure is mentioned in [7], a mapping of the units and connections to a standard middleware like YARP or ROS seems straightforward.

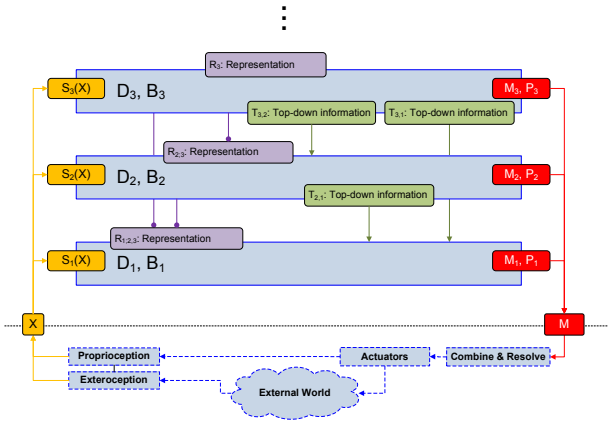

Figure 5: Schematic view of the SYSTEMATICA formalism. Taken from [8].

B5 (?) Design units translate to implementation units, missing interfaces and implied dependencies make reuse of subsystems unpredictable.

4.4. SYSTEMATICA

The SYSTEMATICA formalism introduced by Goerick[8] aims at providing a uniform description language for hierarchical system architectures. It takes the idea of incremental system layers (as also found in Subsumption[12]) and adds bottom-up representation and top-down modulation channels, see Fig. 5.

Our evaluation of the **SYSTEMATICA** formalism thus looks as follows:

A1,2 (?) The formalism decomposes systems into units, but each of these units is required to present a full sensor-motor loop. This rough and one-dimensional description allows the expression of arbitrary systems, but not in arbitrary detail.

A3 (+) The formalism itself is mathematically formalized.

A4 (+) Interfaces are specified by representation and top-down output, input ports are implied.

B1 (–) The constraint of full sensor-motor loops is often too rough for a decomposition to individual scientists.

B2 (+) Coupling and dependencies are specified by the top-down and bottom-up channels.

B3,4 (+) A translation to the ToolBOS[30] infrastructure is presented in [19] and allows partial execution along the bottom-up / top-down dependencies[31].

B5 (?) Design units translate to implementation units, rough decomposition makes reuse without modification difficult.

4.5. Conclusion

The evaluation of the four discussed formalisms reveals different benefits and drawbacks. What strikes out is that most approaches are not designed to allow an easy implementation of the described EI system, mainly for one of three reasons: rough description granularity (3-Tier / SYSTEMATICA), missing description of interfaces and dependencies (CogAff) or lack of standardization (Boxes and Arrows). On the other hand, as discussed in section 3.3, software infrastructures are not able to capture or design the functional aspects of these systems, like communication patterns, incremental representations, etc. Finally, established approaches from software engineering to remedy these issues, such as UML or xADL, require a high level of precision and predefinition in all elements of the design, which is not suitable for the dynamic process of system integration in EI research (see section 3.1).

We believe that the formulated criteria allow to judge how well a given design language finds a compromise between these three poles: by asking for a flexible and meaningful description (functional design) in parallel to the ability to map to and exploit software infrastructure as well as define interfaces, dependencies and decomposition to individuals (research flexibility).

5. SYSTEMATICA 2D

So far we have formulated and applied a set of criteria for system integration formalisms. Although we hope that the introduced measure is applicable and useful beyond this work, this is not essential: we can already pinpoint the merits and drawbacks of the evaluated existing formalisms: Boxes and arrows are very flexible, but neither implementation-oriented nor standardized, 3-Tier is very focused on implementation but too rough to describe collaboration, CogAff allows a flexible and implementable organization of units but without interfaces or behavior space description which in turn is supported by SYSTEMATICA, which is again too rough and enforces a one-dimensional organization. In order to evolve a new way of writing systems we want to combine the flexibility of boxes and arrows with the implementation- and collaboration-orientation of CogAff and SYSTEMATICA.

The result is the development and formalization of 'SYSTEMATICA 2D' (short: 'SYS2D'). It describes systems on two levels: the functional and the descriptive. In set notation, a system $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$ is defined in SYS2D by a set of functional units $\boldsymbol{U}$, including interfaces, connections and dependencies, and a set of sub-architectures $\boldsymbol{A}$, including the description of their context, semantics and sensory & behavior spaces. Fig. 6 shows an example design, all relevant elements will be detailed in the following.

5.1. Functional System Design

On the functional level, a SYS2D system is composed of $\boldsymbol{N > 2}$ processing units $\boldsymbol{u_n \in U, n = 1..N}$. There is always one unit $\boldsymbol{u_1}$ is responsible for emitting sensor events from exteroception $\boldsymbol{S_e}$ and proprioception $\boldsymbol{S_p}$ as the full sensor space $\boldsymbol{S = S_e \times S_p}$. A second predefined unit $\boldsymbol{u_2}$ is responsible for receiving and executing motor commands from the motor space $\boldsymbol{M}$.

5.1.1. Formal Notation

Every unit $\boldsymbol{u_n = \{(D_n, I_n, O_n, \mathbf{Pull}_n, \mathbf{Push}_n)\}}$, $\boldsymbol{n = 1..N}$ is described by the following features (see Fig. 6):

- it has an internal dynamics $\boldsymbol{D_n}$ running independently and asynchronously from all other units;
- it has an interface defined by a set of input ports $\boldsymbol{I_n}$, where each element is defined by its name, type and input role, thus $\boldsymbol{I_n \subset \{(\mathbf{name}, \mathbf{type}, \mathbf{role})\}}$ and a set of output ports $\boldsymbol{O_n}$, where each element is defined by its name and type, thus $\boldsymbol{O_n \subset \{(\mathbf{name}, \mathbf{type})\}}$;
- it specifies the properties of each input port by assigning one of three 'roles', which we will call **Driving**, **DrivingOptional** or **Modulatory** —these roles define dependencies between units as will be detailed in section 5.1.2;
- it may pull data from another units output port $\boldsymbol{o'}$ to one of its input ports $\boldsymbol{i}$, specified by a set of pull operations $\boldsymbol{\mathbf{Pull}_n \subset \{(u_{\text{source}}, o', i)\}}$;
- it may push data from one of its output ports $\boldsymbol{o}$ to another units input port $\boldsymbol{i'}$, specified by a set of push operations $\boldsymbol{\mathbf{Push}_n \subset \{(u_{\text{target}}, i', o)\}}$.

A description of a system as a set of units (with arbitrary granularity), communicating over arbitrary connections is intuitively very flexible but does not enforce consideration of unit dependencies, sub-system reusability or infrastructure exploitation. Dependencies, and thereby structural bias, are expressed by two means: connections can be formulated symmetrically as pull or push and each input port has a specific role. We will discuss these language elements in the following before deriving system properties and constraints, like incremental construction and global deadlock-free operation, in section 5.4.

5.1.2. Input Roles & Dependencies

Two formal elements allow formulating dependencies: push/pull connections and input roles. These two mechanisms are independent and can therefore be used to specify dependencies along two independent dimensions: the difference between push/pull defines the 'build order' dimension, the roles of input ports define the 'processing flow' dimension.

Build Order: If unit $\boldsymbol{u_n}$ pulls data from or pushes data to unit $\boldsymbol{u_m}$ then $\boldsymbol{u_n}$ has to be built after unit $\boldsymbol{u_m}$ — in other words: only the newer unit needs to know about the *connections* it makes to older or preexisting units (although the older units must provide the *ports* to accept

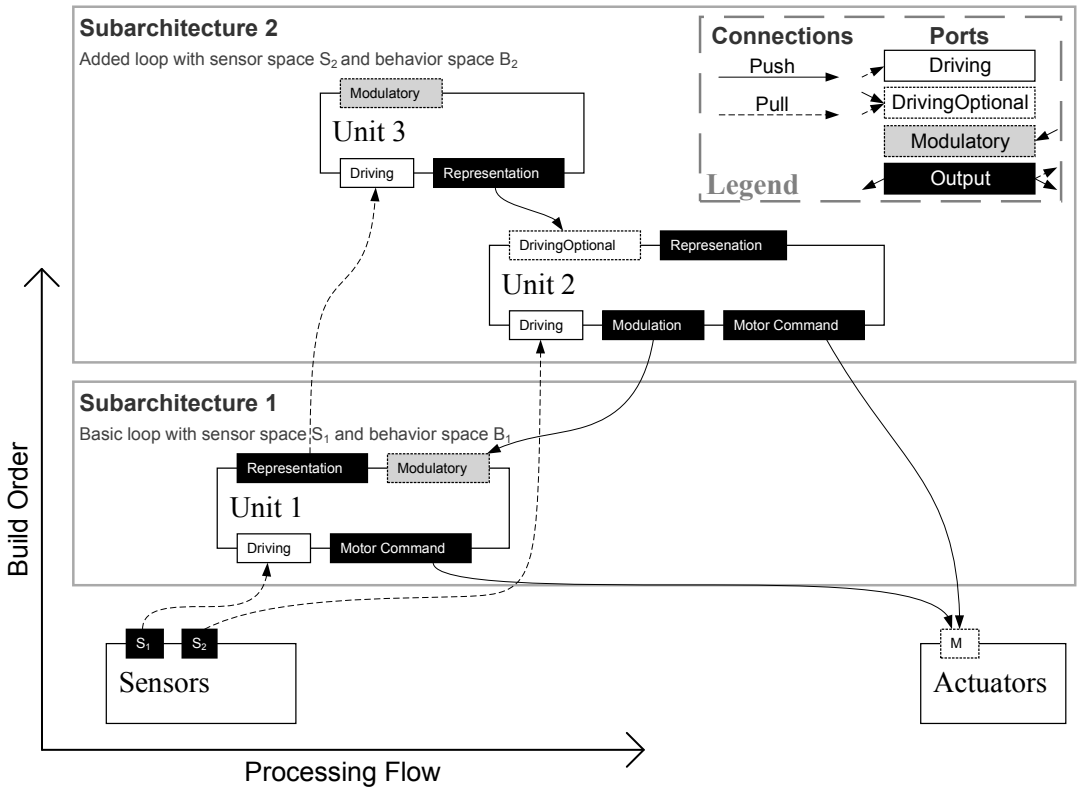

Figure 6: Visualization of a SYSTEMATICA 2D description. The system is composed of units which are arranged along the two axis according to processing flow and build order. Black ports are outputs, light gray ports are Modulatory inputs, white ports with solid line are Driving inputs and white ports with dashed lines are DrivingOptional inputs.

these connections). Since every connection between ports can be symmetrically formulated as either a push or a pull, this sorting by build order is completely in the hands of the designer.

Processing Flow The concept of sorting units by their role or function in the processing chain is old: from the Sense-Plan-Act models, over the Controller-Sequencer-Deliberator sorting in 3-Tier to the Bottom-Up and Top-Down channels in SYSTEMATICA—not to mention the frequent usage of these terms in neurological studies.

In the SYS2D functional model, we chose to model this quality locally, by specifying the 'role' of input ports as one of the following three (see Fig. 7):

- **Driving** inputs are mandatory and indicate input data from units prior to the recipient along the processing flow —this is typically used for sensor preprocessing results, representations, etc.
- **DrivingOptional** inputs are similar to Driving but optional, i. e. the recipient can function without receiving data on such ports —this is typically used for inputs to data fusion units, motor commands, etc.
- **Modulatory** inputs are optional and indicate input data from units further along the processing flow — this is typically used for modulation of parameters or operation modes

One combination is intentionally missing: mandatory inputs from modules further along the processing flow (i. e. mandatory modulation). This is perhaps the strongest structural bias enforced by SYSTEMATICA 2D; the main motivation is measure criterion B5: if units can form mandatory connections to modulation sources, a decomposition into independent subsystems is impossible. This constraint does not prohibit processing loops in a system but only requires some links in a processing loop to be declared as loosely coupled, i. e. DrivingOptional or Modulatory.

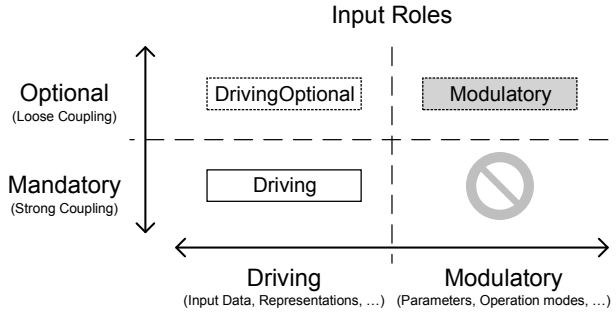

Figure 7: Input roles in SYSTEMATICA 2D. Two criteria are interleaved: driving / modulatory inputs (sometimes referred to as bottom-up / top-down) and mandatory / optional inputs. Three of these combinations are supported by the designated roles, the combination 'mandatory and modulatory' is excluded by design. See text for details.

Specific examples of the impact of this constraint will be discussed in section 7.2.

Sorting along the processing flow is now straightforward. If unit $\boldsymbol{u_n}$ receives (by push or pull) data to a Driving or DrivingOptional input from unit $\boldsymbol{u_m}$ then unit $\boldsymbol{u_n}$ is further along the processing flow than unit $\boldsymbol{u_m}$. Conversely, if unit $\boldsymbol{u_n}$ receives data to a Modulatory input from unit $\boldsymbol{u_m}$ then unit $\boldsymbol{u_m}$ is further along the processing flow than unit $\boldsymbol{u_n}$.

5.2. Functional vs. Technical Aspects

Both dimensions, build order and processing flow, could be interpreted as purely technical categorizations to improve implementation. However, from a technical point of view, there is no important difference between DrivingOptional and Modulatory inputs (both are optional or 'loosely coupled'). Even the build order would be superfluous since relations like 'build A before B' can be derived directly from the dependencies defined by input roles.

Our motivation for distinguishing push/pull connections and DrivingOptional/Modulatory inputs is therefore much more motivated by the goal of establishing a functional relation between units in addition to the goal of using functional design elements purely for a technical implementation. The separation of DrivingOptional and Modulatory inputs follows the distinction between sensor-near to sensor-far data flow and vice versa, thus defining a dimension from sensor to internal representation to actuator ('processing flow'). The separation of push and pull connections allows sub-system separation in a much stronger way than by the unit dependencies alone, namely into incremental construction blocks (definition follows in section 5.4) similar to the phylogenetic evolution of the control structure of a biological organism. A discussion of the relation between functional behavior of a system and the positioning of units in these two dimensions will be done in section 7.3.

5.3. Descriptive System Design

In addition to the set of units, a definition of sensor and behavior spaces is important for understanding and comparing system hypotheses.

We follow the understanding and motivation presented in [8]: The sensor space of a unit or set of units describes which subset of sensors (proprioceptive or exteroceptive) is accessible, the behavior space describes which range of behaviors can be controlled (either by direct motor commands or by modulation of other units). This allows understanding which sub-systems have access to specific (e. g. visual) sensors and which sub-systems are able to trigger specific externally visible behaviors.

Since this is an independent level of description whose granularity may not coincide with that of the units, SYSTEMATICA 2D allows description of sensor and behavior spaces in what we will call 'sub-architectures', composed of one or more units. In this way, the definition of descriptive elements does not impose constraints on the granularity of the functional decomposition.

In a SYS2D design $\mathbb{S} = \boldsymbol{\{U, A\}}$, a sub-architecture $\boldsymbol{a_k} \in \boldsymbol{A}$ is a tuple $\boldsymbol{a_k} = (\textbf{name}, \boldsymbol{U_k}, \boldsymbol{S_k}, \boldsymbol{B_k})$ with $\boldsymbol{U_k} \subset \boldsymbol{U}$ and $\forall \boldsymbol{(k,l)} : \boldsymbol{U_k \cap U_l = \emptyset}$ (a unit may not belong to more than one sub-architecture), where $\boldsymbol{S_k}$ describes the sensor space used by $\boldsymbol{a_k}$ and $\boldsymbol{B_k}$ describes the behavior spaces emitted by $\boldsymbol{a_k}$ (see Fig. 6 for a complete SYS2D design).

5.4. Functional Constraints

Several system properties can be ensured by formulating constraints on the functional side of a SYS2D design (the set $\boldsymbol{U}$):

1. Sortability along the build order (vertical) and processing flow (horizontal) dimensions,
2. Ability for incremental construction,
3. Completeness of a subgraph in terms of necessary units to run the subgraph,
4. Global deadlock-free operation.

Sortability requires that the sorting relations be free of loops in both dimensions: there should be no loops based on push/pull connections as well as no loops based on connections to driving or modulatory inputs. Two examples of the kinds of design this affects can be found in the discussion (see section 7.2).

Definition 5.1 (Sorting Relations). *A unit $\boldsymbol{u_n} \in \boldsymbol{U}$ will be called* **'left of'** *$\boldsymbol{u_m} \in \boldsymbol{U}$ ($\boldsymbol{u_n} <_{\boldsymbol{h}} \boldsymbol{u_m}$) iff there is a push or pull from an output of $\boldsymbol{u_n}$ to a Driving or DrivingOptional input of $\boldsymbol{u_m}$ or there is a push or pull from an output of $\boldsymbol{u_m}$ to a Modulatory input of $\boldsymbol{u_n}$. A unit $\boldsymbol{u_n} \in \boldsymbol{U}$ will be called* **'below'** *$\boldsymbol{u_m} \in \boldsymbol{U}$ ($\boldsymbol{u_n} <_{\boldsymbol{v}} \boldsymbol{u_m}$) iff there is a push $\boldsymbol{p} \in Push_{\boldsymbol{m}}$ connecting an output of $\boldsymbol{u_m}$ to an input of $\boldsymbol{u_n}$ or there is a pull $\boldsymbol{p'} \in Pull_{\boldsymbol{m}}$ connecting an input of $\boldsymbol{u_m}$ to an output of $\boldsymbol{u_n}$. To detect loops in these relations, we compute the transitive hulls $<_{\boldsymbol{v}}^{+}$ and $<_{\boldsymbol{v}}^{+}$. The transitive hull of a relation $<$ is the smallest relation $<^{+}$ which contains all elements of $<$ and is transitive, i. e. if $\boldsymbol{a} <^{+} \boldsymbol{b}$ and $\boldsymbol{b} <^{+} \boldsymbol{c}$ then also $\boldsymbol{a} <^{+} \boldsymbol{c}$; it can be computed by starting with $<^{+}=<$ and incrementally finding triples $\boldsymbol{(a,b,c)}$ and adding $\boldsymbol{(a,c)}$ to $<^{+}$.*

Proposition 5.1 (Sortability). *A system $\mathbb{S} = \boldsymbol{(U, A)}$ is* **sortable** *iff the relations $<_{\boldsymbol{h}}^{+}$ and $<_{\boldsymbol{v}}^{+}$ are antisymmetric, i. e. if there is no pair $\boldsymbol{(a,b)}$ with $\boldsymbol{a} <^{+} \boldsymbol{b}$ and $\boldsymbol{b} <^{+} \boldsymbol{a}$.*

PROOF. Since the sorting relations are defined independently, sorting in horizontal and vertical direction can also be done independently and equivalently with $<^{+}$ representing $<_{\boldsymbol{h}}^{+}$ and $<_{\boldsymbol{v}}^{+}$. The weakest form of sorting, i. e. the ability to assign order numbers $\boldsymbol{o_n} \in \mathbb{N}$ to all units $\boldsymbol{u_n} \in \boldsymbol{U}$ so that the order obeys the sorting relation

($\boldsymbol{u_n} <^+ \boldsymbol{u_m} \Rightarrow \boldsymbol{o_n} < \boldsymbol{o_m}$), is provided by partially ordered sets[32], which are defined over relations which are transitive and antisymmetric. Since $<^+$ is transitive and antisymmetric it can be used as ordering relation in the partially ordered set $\boldsymbol{(U, <^+)}$, thus making the set $\boldsymbol{U}$ sortable (along the horizontal and vertical axes independently). □

Several newer works define partially ordered sets on $\leq$ instead of $<$ relations and require them to be reflexive. The sorting relations introduced here do not define 'equality' between units, thus the $<$-sign was used; this also means that there is no necessity for the relations $<_h^+$ and $<_v^+$ to be reflexive as it is not an essential precondition for sortability of a partially ordered set[32].

Incremental construction, as the second property to be evaluated, requires that mandatory, i. e. Driving, inputs are connected and come from preexisting, i. e. older, units.

Definition 5.2 (Incremental Construction). *A system* $\mathbb{S} = \boldsymbol{(U, A)}$ *is said to* **allow incremental construction** *iff it is sortable and for every unit* $\boldsymbol{u_n}$*, for every Driving input port* $\boldsymbol{(i, t, Driving)} \in \boldsymbol{I_n}$ *there is a pull connection* $\boldsymbol{(u_m, o, i)} \in \boldsymbol{Pull_n}$ *providing data to that port.*

Proposition 5.2 (Complete Subgraph). *Given a system* $\mathbb{S} = \boldsymbol{(U, A)}$ *which allows incremental construction, a subgraph* $\boldsymbol{U_k} \subset \boldsymbol{U}$ *is executable if all mandatory inputs of all units* $\boldsymbol{u_n} \in \boldsymbol{U_k}$ *receive data from within the subgraph. Thus, the subgraph* $\boldsymbol{U_k}$ *is executable if it contains the full transitive subgraph under* $<_v$ *of each contained units:*

$$\forall(\boldsymbol{u_n} \in \boldsymbol{U_k}, \boldsymbol{u_m} \in \boldsymbol{U}) : \boldsymbol{u_m} <_v^+ \boldsymbol{u_n} \Rightarrow \boldsymbol{u_m} \in \boldsymbol{U_k}$$

Proof. Incremental construction requires all Driving input ports to receive data by pull connections. A pull connection between two units implies that the two units are related with $<_v$. By transitive completion from $<_v$ to $<_v^+$, each unit is related to all units required to provide all Driving and therefore all mandatory inputs. □

Proposition 5.3 (Global deadlock-free operation). *Every system* $\mathbb{S} = \boldsymbol{(U, A)}$ *which is composed of local deadlock-free units and is sortable is global deadlock-free.*

Proof. In any system of locally deadlock-free, asynchronously running units, global deadlocks can occur only by subsets of units waiting on each other because of loops of mandatory inputs. Mandatory input is only permitted using Driving inputs, therefore all connections (both push and pull) to a mandatory input from sending unit $\boldsymbol{u_n}$ to receiving unit $\boldsymbol{u_m}$ imply $\boldsymbol{u_n} <_h \boldsymbol{u_m}$. Since the system is to be sortable, the transitive hull $<_h^+$ of $<_h$ is required to be antisymmetric, i. e. there is no pair $\boldsymbol{(u_n, u_m)}$ with $\boldsymbol{u_n} <_h^+ \boldsymbol{u_m}$ and $\boldsymbol{u_m} <_h^+ \boldsymbol{u_n}$. Therefore, the system cannot contain loops of mandatory inputs. □

Definition 5.3 (Valid Sys2d Design). *A Sys2d system* $\mathbb{S} = \boldsymbol{(U, A)}$ *is said to be* **valid** *iff it is sortable and allows incremental construction.*

The term 'valid Sys2d design' thus combines all formulated constraints (enforcing the highest level of structural bias possible with Systematica 2d) and all derived benefits (incremental construction, sub-system decomposition and global deadlock-free operation). All future discussions concerning the impact of structural bias and the benefits of Systematica 2d in general will concentrate on 'valid' Sys2d designs.

5.5. Visual Representation

To allow faster understanding and communication, a visual representation of the Sys2d description, so far described in set notation, is clearly preferable. We restrict this representation to sortable Sys2d systems to allow the distributions of units along the horizontal and vertical axis to be exploited visually. Fig. 6 shows a graph of such a system. Units are shown as large boxes with their input and output ports arranged on the top and bottom sides. Sub-architectures are shown as containers around sets of units, adding a name and descriptive properties. Ports are colored as follows:

- **Driving** inputs are white with black, solid boundary,
- **DrivingOptional** inputs are white with black, dashed boundary,
- **Modulatory** inputs are light gray and
- **Outputs** are black with white text.

Ports are sorted and assigned to unit top or bottom side so that crossing connections are minimized, but this has no conceptual meaning. Since the depicted systems are sortable, every connection is initiated by the higher unit along the Y axis: connections from a low output to a high input are 'pulled' by the higher unit, connections from a high output to a low input are 'pushed' by the higher unit. Thus, the visual representation can cover all properties of sortable Sys2d systems. Based on this visual representation, a visual editing software was created, please see Appendix.

6. Evaluation

After the definition of the Systematica 2d system design and description language, the most pressing questions are what we can do with it and why it is better than the existing formalisms evaluated in section 4. We will not present a full system instance designed with Sys2d in order to keep this publication self-contained, please refer to [33] for a detailed description of a large-scale EI system built based on a Sys2d design. Rather, to answer the

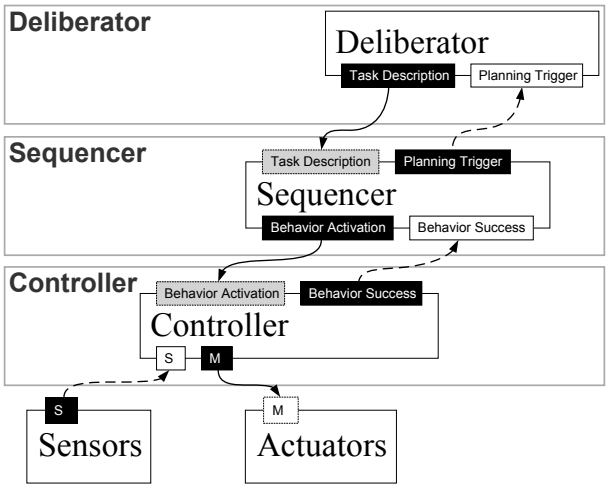

Figure 8: Sys2d visualization of the 3-Tier example from Fig. 3.

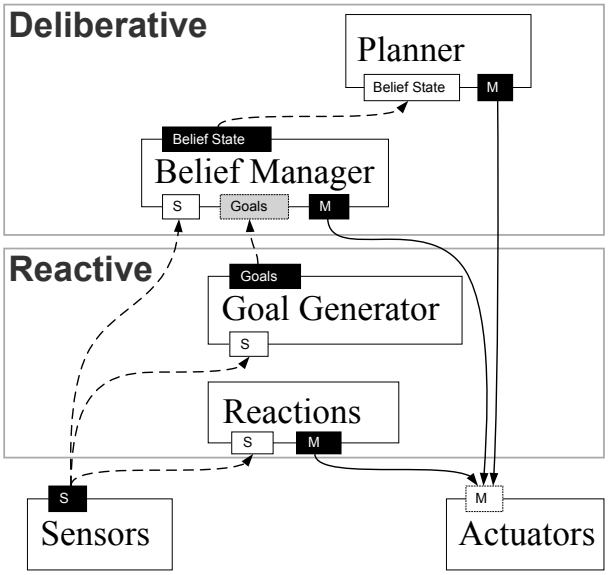

Figure 9: Sys2d visualization of the CogAff example from Fig. 4.

first question, we will present 'translations' of the evaluated formalisms 3-Tier, CogAff and Systematica to the new language. Based on these examples and the functional constraints formulated in section 5.4 we will then evaluate Systematica 2d with the measure criteria presented in section 3.

6.1. Translation of existing formalisms

Figures 8, 9 and 10 show visualizations of Sys2d designs for the 3-Tier, CogAff and Systematica examples, respectively. All three systems are sortable and allow incremental construction; this is not a property of the "translation" but shows that these qualities are important in other notations as well. The visualization can therefore be done along the lines described in section 5.5.

In the process of translating from the original formalism to Systematica 2d several pieces of information had to be added to arrive at a complete system description.

For the 3-Tier case, this is the question of triggered or deliberative planning —a question which is in general undecided in system theory but which still requires a decision for every specific system instance. Fig. 8 shows the case of triggered planning.

For the CogAff example, the interfaces had to be defined in more detail, which was done based on explanations given in [7]; dependencies and input roles were chosen to fit the two-dimensional arrangement already inherent in CogAff.

Finally, for the Systematica example, input ports had to be defined where the original formalism only specifies representations and top-down outputs; because of the one-dimensional nature of Systematica the units are arranged diagonally.

The examples not only show that Systematica 2d is able to express the evaluated formalisms adequately but that it helps to ask questions necessary to complete their notation.

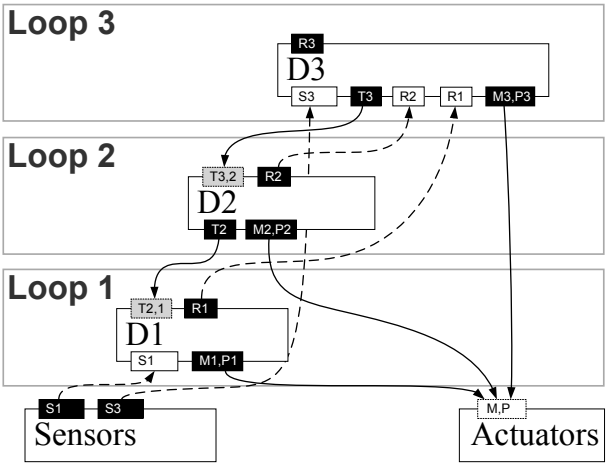

Figure 10: Sys2d visualization of a Systematica example similar to Fig. 5.

6.2. Evaluation of Systematica 2d *in the Measure*

Our evaluation of the Systematica 2d formalism thus looks as follows:

A1 (?) The formalism decomposes systems into units with inputs, outputs and connections. The proposed functional constraints reduce this flexibility slightly, which is a constructive bias for most EI systems but might limit the expressiveness in other domains (see discussion in section 7.2).

A2 (+) The formalism allows a fine granularity in the description and it is therefore up to the designer to choose which level of detail is needed —although it needs to respect criterion B1 (Decomposition to individuals). Beyond the functional units, a description of sub-architecture properties is also possible.

Systematica 2d	XCF	YARP	ROS
Units	Processes	Processes	Processes
Connections (push+pull)	Active Memory (XML-RPC)	YARP ports	ROS (XML-RPC)
Driving	Subscription + sync.	yarp-connect + sync.	Subscription + sync.
DrivingOptional	Query	Observer	Subscription
Modulatory			

Table 1: Mapping of Sys2d functional description elements to existing software infrastructures.

A3 (+) The formalism itself is mathematically formalized.

A4 (+) Interfaces are explicitly described by input and output ports.

B1 (+) The granularity can be matched to a per-scientist decomposition (see A2).

B2 (+) Coupling and dependencies are specified by input roles and push/pull connections.

B3 (+) The formalism can be translated to any infrastructure which supports communicating units and the three used input roles (see [31]) —which is possible in practically every infrastructure, from service-oriented systems over agents and black-board systems to data-flow and OOP-based engines.

B4 (+) The functional properties of incremental construction and complete subsystems defined in section 5.4 aim directly at utilizing infrastructure properties of partial testing and graceful degradation.

B5 (?) The fine decomposition makes reusing single units possible without change (at least when observing B1). In addition, systems fulfilling the incremental construction constraint are very easy to decompose into reusable subsystems. However, a reuse of sub-architectures as 'composite' units in future systems is planned but currently not formalized.

7. Discussion

7.1. Mapping Sys2d *Designs to Software Infrastructure*

Following the discussion about the design power of software infrastructures in section 3.3, the last step in closing the hypothesis test cycle using Systematica 2d is to provide a mapping of the functional elements of a Sys2d design to mechanisms present in specific established software infrastructures. Such a mapping is necessary since even though Sys2d uses terms like 'interfaces', 'asynchronously' running units or 'build order', these are not defined with a particular software engineering concept in mind but in order to provide a formal system description, including the constraints and proofs presented in section 5.4.

Table 1 shows an overview of the proposed mapping of the *technical* design elements found in the functional description of Sys2d designs to XCF, YARP and ROS. These technical elements are units, connections (without the push/pull distinction) and input roles (without the DrivingOptional/Modulatory distinction). Thus the infrastructure has to support asynchronously running processes/threads/components/etc., at least one way to exchange data between them and the ability for tight and loose coupling.

Common to all mappings is that units are to be implemented as separate processes, using the mechanisms provided by the infrastructure to interact. In XCF, this interaction goes thought the Active Memory in order to benefit from the easy addition and removal of processes, as well as from the memory dynamic for modulation signals (e. g. removing a parameter set if it is not regularly updated). In YARP, processes communicate directly (using YARP ports) in an observer pattern, whether a connection is a push or a pull implies which of the processes performs the registration. In ROS, processes also use the communication mechanism provided by the infrastructure which relies completely on subscriptions.

Extra care is necessary for implementing tight and loose coupling since all three infrastructures use subscriptions by default. For Driving inputs, a synchronization to observed / subscribed channels may be necessary, for DrivingOptional and Modulatory inputs it must be ensured that received data is not outdated (i. e. that the sending unit / process is still alive). An algorithm to achieve this loose coupling can be found in [31].

In summary, we believe that mapping Sys2d designs to these (and other) software infrastructures is unproblematic since unit decomposition, interfaces and connections in the design serve not only the functional description but always keep the future implementation in mind.

7.2. Discussion of Structural Bias in Systematica 2d

After presenting the formalism itself and the benefits we believe it entails for research system integration, we will now discuss the specific impact of the implied structural bias and the constraints this imposes for system design. From an ADL point of view, one could say that the Sys2d language provides pre-defined component (units) and connector types (input roles) and uses those to enforce constraints on the explicit architecture configuration chosen by the designer. In this section we will analyze the kinds of configurations favored by these constraints by looking at two technical communication patterns (Client-Server and Publisher-Subscriber) and one popular artificial intelligence design pattern, "Lateral Support". On these examples we will show which kinds of design are favored by Sys2d, mainly by restrictions on input roles and sortability, and why we believe this to be an advantage.

7.2.1. Server-Client

The typical way of connecting server and clients is by every client pushing requests to the server and the

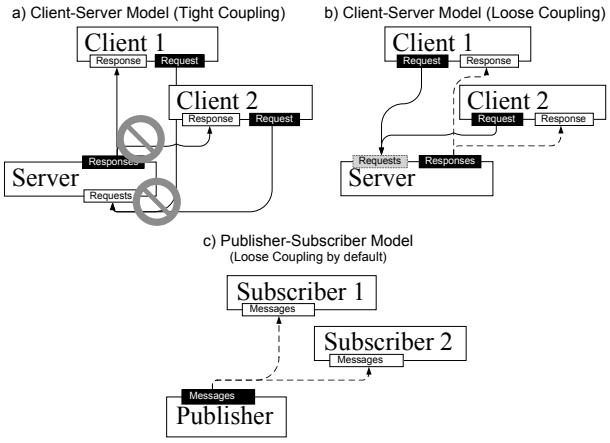

Figure 11: Common interaction scenarios to illustrate and motivate the structural bias of SYSTEMATICA 2D. a) Tight coupled Server-Client layout, where clients push requests and the server pushes responses —this creates a build-order unsortability (a dependency loop) which makes incremental construction impossible. b) Alternative, SYS2D-compatible loose coupled Server-Client layout where clients push requests to the server but pull results back once the server provides them. c) Publisher-Subscriber layout, subscription is modeled by pull connections, thus ensuring separability and sortability.

server pushing responses back to each client —the resulting SYS2D design is depicted in Fig. 11a. We see several drawbacks in this design: First, the two-way push makes it unclear which unit could run without the other: both units depend on each other, which makes independent development or testing difficult (see measure criteria **B1** and **B4** in section 3.4). Second, the use of Driving inputs for both requests and responses (assuming server and client wait for these inputs, which is the typical case) implies synchronization of both partners to each other, thus undermining the idea of asynchronous processing and impairing the ability for subsystem decomposition (criterion **B5**). Both objections are reflected in the SYS2D constraints: the graph shown in Fig. 11a is sortable neither in horizontal nor in vertical direction.

Fig. 11b shows an alternative interpretation of a Server-Client layout which is compatible with the SYS2D constraints. The server is modeled as the base unit, receiving request via an optional, modulatory input and publishing —but not pushing —the results. Clients thus push their requests to this modulatory server input and pull back results from the server. This allows the server to run asynchronously, be tested independently and be separated and reused.

We would like to point out that both designs, the non-compatible and the compatible, have processing loops between server and clients. The structural bias in SYSTEMATICA 2D does not prohibit loops but merely ensures that they are not tightly coupled: in every processing loop there must be at least one connection with loose coupling —this usually requires only a few design adjustments and allows the described benefits.

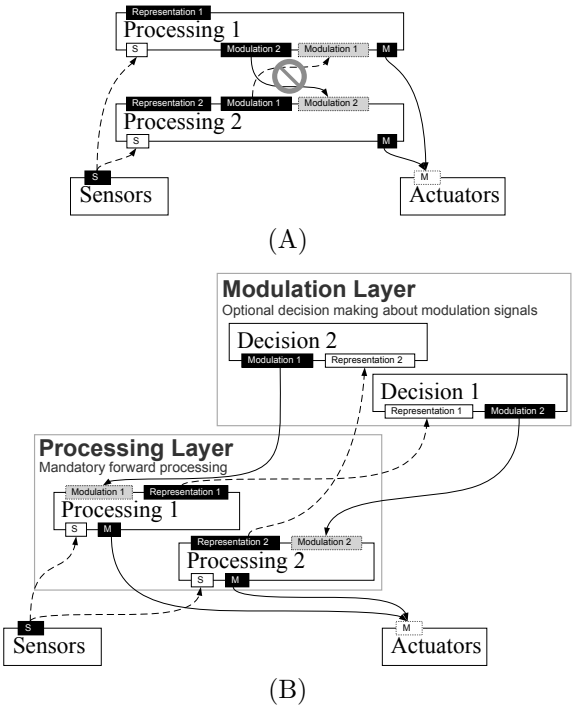

Figure 12: **(A)** Straightforward modeling of the lateral support pattern: the mutual modulation produces a processing flow unsortability. **(B)** Model of lateral support from a SYSTEMATICA 2D point of view. To ensure separability, processing and decision, or support generation, are separated. We see this as a general principle: in order to reduce interconnectivity and improve robustness against failing units, processing units are separate from decision units providing modulation signals.

7.2.2. Publisher-Subscriber

A SYS2D interpretation of this layout can be seen in Fig. 11c: The publisher asynchronously generates messages which are pulled from the clients to their driving inputs. In this interpretation, the actual 'subscription' process is implied in the setup of the pull connection. With this design, the same properties of asynchronous operation, independent testability and separability as described for the Server-Client layout also apply here.

7.2.3. Lateral Support

It is a popular technique to use (intermediate) results of one processing flow, e. g. confidence ranges, to improve processing of a parallel processing flow —this is commonly referred to as lateral support. In a straightforward modeling of two units (see Fig. 12A), mutual modulation leads to a similar problem as discussed for the 'typical' Server-Client layout in section 7.2.1, just that the sortability violation is in the 'processing flow' dimension.

We therefore propose an alternative layout, as shown in Fig. 12B: it is based on the concept that the logic for performing each processing flow and the logic for applying intermediate results as modulation to another processing should be separate. In the proposed design, this second piece of logic is called 'Decision' unit. Using such a processing/decision separation recovers several important

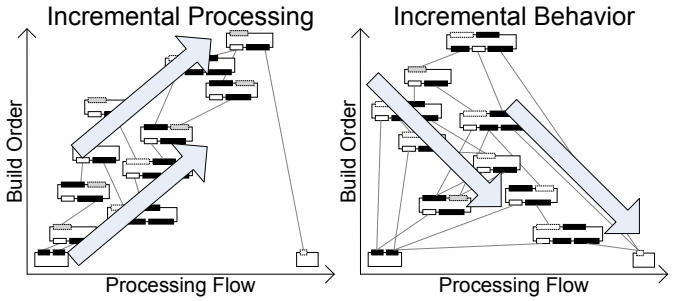

Figure 13: Schematic comparison of system-wide integration approaches. Left: (Pre-)Processing steps are incrementally added until the internal representation supports making complex decisions (reasoning, planning, ...), thus a motor command is produced by the top-most component; Right: Simple behavior generation is built first and then enriched / modulated by more complex preprocessing, decision making and behavior generation, motor commands are produced during all phases of development.

properties: First, the graph is sortable again by avoiding the two-way modulation; by extension it also allows incremental construction. Second, the separation into a basic processing layer and an added decision layer helps with partial testing and ensures that processing can go on if the decision layer fails (graceful degradation). Finally, in case a processing module is to be used in a different context where this form of lateral support is not possible, it can be used without the decision module specific for this purpose (subsystem separation).

7.3. Comparison of Integration Approaches

In addition to the modeling of specific systems, we believe SYSTEMATICA 2D can also be used to relate *integration approaches*, not primarily by their set notation but by the visual representation of the horizontally and vertically arranged units of these sets. It is in the nature of this sorting to show a correlation between build order (vertical) and processing flow from sensor to actor (horizontal). Two schematics of possible arrangements are visualized in Fig. 13, we will call the left 'Incremental Processing' and the right 'Incremental Behavior'.

The 'Incremental Processing' approach starts with sensory (e. g. visual) preprocessing, adds cue fusion or postprocessing, proceeds to scene analysis and reasoning, and finally derives motor commands from planning. Thus, units are arranged along the secondary diagonal, from lower left to upper right and into the top left corner. EI systems based on this approach will require a sophisticated sensory processing for the artifact to do anything at all (the danger being that the integration process gets stuck here), but once that is achieved opens the door to powerful reasoning and planning.

The 'Incremental Behavior' approach starts with very rough (typically multi modal) sensory processing to enable a quick selection of basic behaviors (e. g. approach / retreat) which can already allow behavior learning. More sophisticated preprocessing and sensor fusion is added on top of that, necessitated by more complex tasks and behaviors, resulting in symbolic storage and reasoning when this is actually required for a task. Thus, units are arranged along the primary diagonal, from upper left to lower right and in the lower left corner. EI systems based on this approach will start with behavior generation and learning as the first step (the danger being that the system gets stuck in toy scenarios), but give all further processing layers a behaviorally grounded frame of reference.

It is not our aim to judge in this long dispute, but to show that differences in the integration approach result in very apparent differences in the two-dimensional arrangement of SYSTEMATICA 2D system descriptions.

8. Conclusion

The development, implementation and evolution of architectural structures is essential for the progress of intelligent systems. Formalisms which make comparison to other systems hard, do not consider collaboration sufficiently and in effect lead to long development phases and non-reusable systems are counterproductive for this purpose. We have therefore formulated SYSTEMATICA 2D, a formalism to support this hypothesis test cycle through all three phases: starting with a description of the hypothesis, including functional and descriptive elements, over the crucial support during system integration, based on functional constrains of the design, to the subsystem separation allowed by explicit modeling of dependencies. A measure was introduced to evaluate SYSTEMATICA 2D and identify improvements over existing formalisms.

We have shown that the new formalism is able to express the evaluated existing formalisms and enriches them by explicit interfaces, finer granularity or two-dimensional ordering of units. In addition to the measure, this ability of translation provides another way to relate system integration concepts to one another. The functional design, including the provable properties of incremental construction and global deadlock-free operation, combined with the ability to map to a variety of software infrastructures, makes constructing systems faster. The descriptive design additionally allows analysis of sensor and behavior spaces and relation to other system hypotheses.

Future work will focus on further enriching and applying the new formalism. Planned extensions are the addition of temporal dependencies or latencies as well as a formal description of test-cases for the behavior of units. To allow handling higher numbers of units, a formulation of sub-architectures into 'composite units' will be investigated. However, the main effort of future work with this formalisms will be on translating existing and developing new intelligent system hypotheses as well as categorizing integration approaches in more detail in order to identify the most promising next steps.

Appendix: Visual Editing Software

Based on the visual representation described in section 5.5, a software tool was developed to create, view and edit Sys2d designs. Using the simple shapes of units, ports and sub-architectures, it is possible to quickly combine systems while automatically enforcing the sortability constraint and utilizing it to support the positioning of units. Basis for the tool is an XML format representing the set-notation of Sys2d models, enriched with tags carrying visual data. The tool is platform-independent, allows interactive editing and is able to verify the constraints of *valid* Sys2d designs. Graphs can be exported as PNG or SVG for visual or vector-based post processing, respectively.

To get a copy of the tool, please send a short mail to `benjamin.dittes@honda-ri.de`.

References

[1] D. Vernon, G. Metta, G. Sandini, A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents, IEEE Transactions on Evolutionary Computation 11 (2) (2007) 151–180.

[2] OMG, Unified modeling language (uml) superstructure specification, [Online] (Feb 2009).
URL `http://www.omg.org/spec/UML/2.2/`

[3] N. Hawes, J. Wyatt, Engineering intelligent information-processing systems with cast, Adv. Eng. Inform. 24 (1) (2010) 27–39.

[4] J. Fritsch, S. Wrede, An integration framework for developing interactive robots, Software Engineering for Experimental Robotics (2007) 291–305.

[5] G. Metta, P. Fitzpatrick, L. Natale, Yarp: Yet another robot platform, International Journal on Advanced Robotics Systems 3 (1) (2006) 43–48.

[6] E. Gat, et al., On three-layer architectures, Artificial Intelligence and Mobile Robots.

[7] N. Hawes, Architectures by design: The iterative development of an integrated intelligent agent, in: Proceedings of AI-2009, The Twenty-ninth SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence, 2009.

[8] C. Goerick, Towards an understanding of hierarchical architectures, Transactions on Autonomous Mental Development, Special Issues on Cognitve Architectures to appear.

[9] N. Medvidovic, R. Taylor, A framework for classifying and comparing architecture description languages, Software Engineering - ESEC/FSE'97 (1997) 60–76.

[10] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, W. Mann, Specification and analysis of system architecture using rapide, IEEE Transactions on Software Engineering 21 (4) (1995) 336–354.

[11] D. Garlan, R. Monroe, D. Wile, Acme: An architectural interconnection language, Tech. rep., Technical Report, CMU-CS-95-219, Carnegie Mellon University (1995).

[12] R. Brooks, A robust layered control system for a mobile robot, IEEE journal of robotics and automation 2 (1) (1986) 14–23.

[13] M. Scheutz, J. Kramer, Radic: a generic component for the integration of existing reactive and deliberative layers, in: 5th Intl. joint conf. on Autonomous agents and multiagent systems, ACM New York, NY, USA, 2006, pp. 488–490.

[14] G. Gössler, J. Sifakis, Composition for component-based modeling, in: Sci. Comput. Program., Vol. 55, 2005, pp. 161–183.

[15] G. Abowd, R. Allen, D. Garlan, Formalizing style to understand descriptions of software architecture, ACM Transactions on Software Engineering and Methodology (TOSEM) 4 (4) (1995) 319–364.

[16] A. Sloman, The cognition and affect project: Architectures, architecture-schemas, and the new science of mind.

[17] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, R. Taylor, xadl: enabling architecture-centric tool integration with xml, in: System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on, IEEE, 2002, p. 9.

[18] S. Wrede, J. Fritsch, C. Bauckhage, G. Sagerer, An xml based framework for cognitive vision architectures, in: 17th Intl. Conf. on Pattern Recognition, 2004.

[19] C. Goerick, B. Bolder, H. Janßen, M. Gienger, H. Sugiura, M. Dunn, I. Mikhailova, T. Rodemann, H. Wersing, S. Kirstein, Towards incremental hierarchical behavior generation for humanoids, Intl. Conf. on Humanoids.

[20] D. Kraft, E. Baseski, M. Popovic, A. Batog, A. Kjær-Nielsen, N. Krüger, R. Petrick, C. Geib, N. Pugeault, M. Steedman, et al., Exploration and planning in a three-level cognitive architecture, in: Proceedings of the International Conference on Cognitive Systems (CogSys 2008), Citeseer, 2008.

[21] B. Leibe, N. Cornelis, K. Cornelis, L. Van Gool, Dynamic 3d scene analysis from a moving vehicle, in: Conf. on Computer Vision and Pattern Recognition, 2007.

[22] M. Proetzsch, T. Luksch, K. Berns, Development of complex robotic systems using the behavior-based control architecture ib2c, Robotics and Autonomous Systems.

[23] D. Vernon, G. Metta, G. Sandini, The icub cognitive architecture: Interactive development in a humanoid robot, in: 6th Intl. Conf. on Development and Learning, 2007, pp. 122–127.

[24] R. Ramsin, R. Paige, Process-centered review of object oriented software development methodologies, ACM Computing Surveys (CSUR) 40 (1) (2008) 1–89.

[25] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, B. Álvarez, V 3 cmm: a 3-view component meta-model for model-driven robotic software development, Journal of Software Engineering for Robotics Vol 1, No 1 (2010) 3–17.

[26] H. Martínez-Barberá, D. Herrero-Pérez, Programming multi-robot applications using the thinkingcap-ii java framework, Advanced Engineering Informatics 24 (1) (2010) 62–75.

[27] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng, Ros: an open-source robot operating system, in: International Conference on Robotics and Automation, 2009.

[28] N. Ando, T. Suehiro, T. Kotoku, A software platform for component based rt-system development: Openrtm-aist, Simulation, Modeling, and Programming for Autonomous Robots (2008) 87–98.

[29] S. Chernova, R. Arkin, From deliberative to routine behaviors: a cognitively inspired action-selection mechanism for routine behavior capture, Adaptive Behavior 15 (2) (2007) 199.

[30] A. Ceravola, M. Stein, C. Goerick, Researching and developing a real-time infrastructure for intelligent systems - evolution of an integrated approach, Robotics and Autonomous Systems 56 (1) (2007) 14–28.

[31] B. Dittes, M. Heracles, T. Michalke, R. Kastner, A. Gepperth, J. Fritsch, C. Goerick, A hierarchical system integration approach with application to visual scene exploration for driver assistance, in: Proceedings of the 7th International Conference on Computer Vision, Springer-Verlag New York Inc, 2009, pp. 255–264.

[32] B. Dushnik, E. Miller, Partially ordered sets, American journal of mathematics 63 (3) (1941) 600–610.

[33] J. Schmuedderich, N. Einecke, S. Hasler, A. Gepperth, B. Bolder, R. Kastner, M. Franzius, S. Rebhan, B. Dittes, H. Wersing, J. Eggert, J. Fritsch, C. Goerick, System approach for multi-purpose representations of traffic scene elements, in: 13th International IEEE Conference on Intelligent Transportation Systems, 2010.