Formal System Design for Intelligent Artifacts

Benjamin F. Dittes

Der Technischen Fakultät der Universität Bielefeld vorgelegt zur Erlangung des akademischen Grades Doktor der Ingenieurwissenschaften.

Juni 2011

Dissertation zur Erlangung des akademischen Grades Doktor der Ingenieurwissenschaften (Dr.-Ing.)

Der Technischen Fakultät der Universität Bielefeld

- am 28.06.2011 vorgelegt von Benjamin Dittes,
- am 19.12.2011 verteidigt und
- am 28.09.2012 genehmigt.

Betreuer:

- Dr. Christian Goerick,
  Honda Research Institute Europe GmbH
- Dr. Sven Wachsmuth,
  Technische Fakultät der Universität Bielefeld

Gutachter:

- Dr. Christian Goerick,
  Honda Research Institute Europe GmbH
- Dr. Sven Wachsmuth,
  Technische Fakultät der Universität Bielefeld

weitere Mitglieder des Prüfungsausschusses:

- Prof. Dr.-Ing. Ulrich Rückert,
  Technische Fakultät der Universität Bielefeld
- Dr.-Ing. Sebastian Wrede,
  Technische Fakultät der Universität Bielefeld

Gedruckt auf alterungsbeständigem Papier nach DIN ISO 9706.

Abstract

This work deals with conceptual and software aspects (i.e. algorithms and structure) of intelligent systems interacting with the real-world. Typical domains for such systems are robotics as well as personal digital- and driver assistance. In decades of research on intelligent systems, a large number of system structures or architectures for intelligent artifacts have been proposed and implemented. However, no established and broadly accepted hypothesis for such a system structure has emerged because no common language or common understanding of the space of architectures exists. This in turn makes scientific discourse about architectures difficult.

In this thesis we aim to improve the process and tools for describing, constructing and evolving the architecture and software of large-scale systems for intelligent artifacts. At the heart of this improvement is the proposed formalism 'SYSTEMATICA 2D', suitable for both flexible description of system architectures as well as for 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.

The new language is shown to find a good compromise between cognitive description, high flexibility and easy implementation. We present ways to map resulting designs to the most popular infrastructure paradigms and derive mathematically provable benefits for the system construction process: incremental composition, graceful degradation, subsystem separation and global deadlock-free operation.

Finally, the powers of the formalism for architecture categorization and comparison are explored. It is analytically shown that there is a direct relation between sensor / behavior spaces (a descriptive design property) and the interfaces and connections of units (a functional design property).

Without lack of generality, examples and results are obtained from two specific, recent and state-of-the-art large-scale systems: ALIS3[1] and AutoSys[2]. Experimental results show that a) modeling a wide variety of systems as SYSTEMATICA 2D designs is possible, b) implementing systems according to such a design is dramatically faster and produces inherent, provable system properties and c) different systems can be related and classified based on the designs.

Contents

List of Symbols

Systematica 2D system

$\mathbb{S}$	A Systematica 2D system, $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$
$\boldsymbol{U}$	The set of units $(\boldsymbol{u_1}, \boldsymbol{u_2}, \ldots \boldsymbol{u_N})$ in $\mathbb{S}$
$\boldsymbol{A}$	The set of sub-architectures $(\boldsymbol{a_1}, \ldots \boldsymbol{a_L})$ in $\mathbb{S}$
$\boldsymbol{S}$	The full vector space of sensors available to the system, composed of exteroception and proprioception: $\boldsymbol{S} = \boldsymbol{S_e} \times \boldsymbol{S_p}$
$\boldsymbol{M}$	The full vector space of motor commands available to the system

Systematica 2D units

$\boldsymbol{u_1}$	The first unit $\boldsymbol{u_1} \in \boldsymbol{U}$, emitting sensor events from $\boldsymbol{S}$
$\boldsymbol{u_2}$	The final unit $\boldsymbol{u_2} \in \boldsymbol{U}$, receiving and executing motor commands from $\boldsymbol{M}$
$\boldsymbol{u_i}, \boldsymbol{u_j}, \boldsymbol{u_n}$	Arbitrary units $\boldsymbol{u_i}, \boldsymbol{u_j}, \boldsymbol{u_n} \in \boldsymbol{U}$
$<_{\boldsymbol{h}}, <_{\boldsymbol{v}}$	Binary horizontal and vertical relations imposing a partial order on the set of units $\boldsymbol{U}$
$<_{\boldsymbol{h}}^{+}, <_{\boldsymbol{v}}^{+}$	Transitive hull of corresponding binary relations $<_{\boldsymbol{h}}$ and $<_{\boldsymbol{v}}$
$\boldsymbol{D_i}$	Internal dynamics of unit $\boldsymbol{u_i}$
$\boldsymbol{I_i}$	Set of inputs of unit $\boldsymbol{u_i}$
$\boldsymbol{O_i}$	Set of outputs of unit $\boldsymbol{u_i}$
$\mathbf{Pull}_{\boldsymbol{i}}$	Set of pulled connections of unit $\boldsymbol{u_i}$
$\mathbf{Push}_{\boldsymbol{i}}$	Set of pushed connections of unit $\boldsymbol{u_i}$

SYSTEMATICA 2D sub-architectures

$\boldsymbol{a_k}$	An arbitrary sub-architecture $\boldsymbol{a_k} \in \boldsymbol{A}$
$\boldsymbol{U_k}$	Set of units contained in sub-architecture $\boldsymbol{a_k}$
$\boldsymbol{S_k}$	Sensor space of sub-architecture $\boldsymbol{a_k}$
$\hat{\boldsymbol{S}}_{\boldsymbol{k}}$	Full sensor space available to $\boldsymbol{a_k}$ either directly or through other units
$\boldsymbol{B_k}$	Behavior space of sub-architecture $\boldsymbol{a_k}$
$\hat{\boldsymbol{B}}_{\boldsymbol{k}}$	Full behavior space available to $\boldsymbol{a_k}$ either directly or through other units
$\boldsymbol{I_k}$	Set of inputs of units $\boldsymbol{u_i} \in \boldsymbol{U_k}$ pulling data from $\boldsymbol{u_1}$
$\boldsymbol{R_k}$	Set of inputs of units $\boldsymbol{u_i} \in \boldsymbol{U_k}$ pulling data from $\boldsymbol{u_j} \in \boldsymbol{U} \setminus \{\boldsymbol{u_1}, \boldsymbol{u_2}\}$
$\boldsymbol{O_k}$	Set of outputs of units $\boldsymbol{u_i} \in \boldsymbol{U_k}$ pushing data to $\boldsymbol{u_2}$
$\boldsymbol{M_k}$	Set of outputs of units $\boldsymbol{u_i} \in \boldsymbol{U_k}$ pushing data to $\boldsymbol{u_j} \in \boldsymbol{U} \setminus \{\boldsymbol{u_1}, \boldsymbol{u_2}\}$

AutoSys domain

$\boldsymbol{GT(t)}$	Set of ground truth regions of interest $\boldsymbol{gt_i(t)} \in \boldsymbol{GT(t)}$ at time $\boldsymbol{t}$
$\boldsymbol{V(t)}$	Set of visual detections at time $\boldsymbol{t}$
$\boldsymbol{GT^*(t)}$	Set of ground truth regions matched by a detection at time $\boldsymbol{t}$
$\boldsymbol{V^*(t)}$	Set of visual detections matched by a ground truth region at time $\boldsymbol{t}$
$\boldsymbol{q(t)}$	Quality function derived from ground truth regions and detections at time $\boldsymbol{t}$

Miscellaneous

$\boldsymbol{N(\mu, \sigma)}$	Normal distribution with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$
$\boldsymbol{t}$	Time
$\boldsymbol{\nu}$	Scalar parameter

List of Figures

List of Tables

1 Introduction

The goal to create complex intelligent artifacts which are real-world capable, practical and useful is as old as the study of intelligence itself. In this effort, a large number of different disciplines is involved, including neuroscience, hardware research, statistical data analysis and evolutionary intelligence studies, to name a few. In this work we will focus on the conceptual and software aspect (i. e. algorithms and structure) of intelligent systems installed on a physical platform and interacting with the real world. Typical domains for such systems are robotics as well as personal digital- and driver assistance systems. We will refer to this specific sub-area of intelligent systems as 'Embedded Intelligence' (EI), with the goal of synthesizing intelligence in the hull of a physical artifact.

Research on EI systems is characterized by two issues: First, all processing must be real-world capable, i. e. it must be able to deal with adequate levels of noise and run at speeds which allow interaction with the world. Second, in order to create a 'complete' real-world artifact, an interplay of a great number of disciplines is required. Although the actual organization of subsystems in any implemented system may vary, they can be roughly categorized into designing or learning of functionalities for

I Information Acquisition from the world,

II Internal Information Processing,

III Action Generation in the world,

IV Subsystem Integration to create a desired overall system behavior and

V Embedding in a physical artifact.

Creating such complete systems is beneficial for studying associations between perception and behavior (e. g. 'grounding'[6]), testing the feasibility of processing algorithms (areas **I-III**) for real-world artifacts, developing the ability to interact with humans[7] and many others. To this end, excellent processing algorithms are an important prerequisite, but whether their interaction creates an intelligent system strongly depends on the system structure. In other words: Intelligence is not the result of individually superior processing algorithms or hardware components but arises from the structure and interaction in an integrated system (see, e. g. [7, 8, 9, 10]).

In decades of research on intelligent systems, a large number of system structures for intelligent artifacts (we will refer to these as 'hypotheses') have been proposed and implemented, starting with the Sense-Plan-Act layout[11], over Subsumption[11] and 3-Tier[3, 12] to CogAff[4] or multi-agent systems[13, 14]. However, none of these has emerged as an established and broadly accepted hypothesis.

The central problem in this search, as Christensen et al. precisely point out, is that

> "[...] there is no agreement on what the space of possible architectures is like, nor on the terminology for describing architectures or on criteria for evaluating and comparing them." [15]

This implies that scientific work on system architectures is difficult because systematic comparison and knowledge transfer are nearly impossible. However, it also allows to derive the necessary phases for devising a system architecture or hypothesis: the first phase is *hypothesis formulation*, i. e. description in a suitable language, followed by *hypothesis evaluation*, usually by *system construction* to allow experiments, and *hypothesis evolution*, potentially based on a *formal comparison* to other hypotheses and resulting in the formulation of a new system hypothesis. We will refer to these phases as the 'Hypothesis Test Cycle'. Improving the search for viable intelligent artifact hypotheses thus implies improving or enabling each of these phases.

Several scientific fields deal with system design and construction, of which the following three are the most relevant (see Fig. 1.1):

1. **Software Engineering**, providing technical, precise design languages such as UML[16] or xADL[17],
2. **Cognitive Description**, providing theoretical system models in several languages like CogAff[10], 3-Tier[3] or SYSTEMATICA [5] and
3. **Software Infrastructure**, providing practical frameworks for collaborative construction of intelligent artifact systems, e. g. ROS[18], XCF[19] or YARP[20].

Since each of these fields has different priorities and different target domains, each field also has different strengths and weaknesses regarding the hypothesis test cycle for intelligent artifacts. Software engineering provides languages to precisely and completely specify large software system, allowing an efficient construction process but occasionally conflicting with the dynamic nature of research system construction. Cognitive descriptions focus on expressing the concept or hypothesis behind an intelligent artifact system, which makes them ideal for theoretic analysis but in many cases provides little support for the construction process. Software infrastructures finally combine all tools necessary for efficient, collaborative system construction but

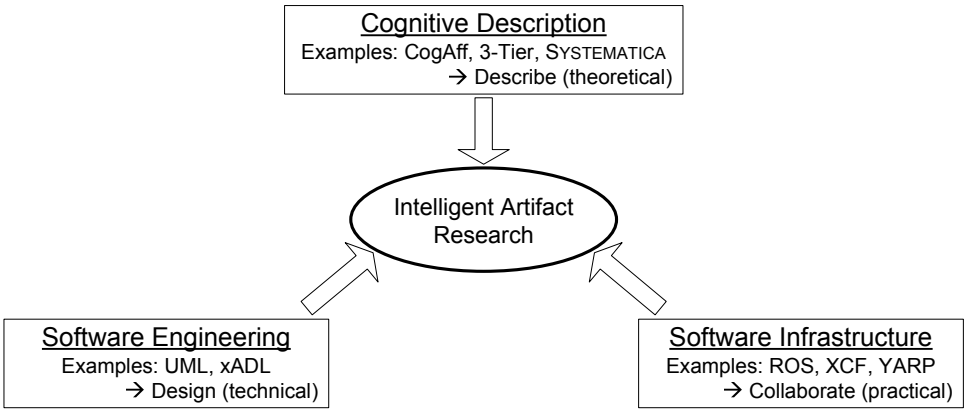

Figure 1.1: Relation of Intelligent Artifact Research Systems to established research directions.

usually lack a design language that allows understanding of component interdependencies or theoretical discourse.

In this work, we will propose a specific formalism, which we will call 'SYSTEMATICA 2D', together with a process to apply it, in order to combine the beneficial properties of these fields specifically for the phases of the hypothesis test cycle: formulating, constructing and evolving architectures for intelligent artifacts. The main contributions of this thesis, i. e. the problems addressed and overcome in each of these three phases, will be detailed in the following.

Hypothesis Formulation

The first step towards any EI system is the formulation, as text, graphical representation or formal expression, of the targeted system's structure. This can be done verbally or implicitly, emerging from the design or implementation of the system on a software infrastructure, but it is nonetheless a form of formulation.

Traditionally, looking for instance at the Sense-Plan-Act models and the later Subsumption[11] and 3-Tier[3] models, the means of formulating systems were linked with a concrete proposal of system structure. This helps in order to tailor the formalisms to the way systems are to be built but allows little generalization of formulation techniques. Only recent models, especially CogAff by Sloman[10] and SYSTEMATICA by Goerick[9] have started to look for underlying patterns of organization in order to establish a notation for various integration approaches and related system hypotheses.

The SYSTEMATICA 2D language introduced in this work will allow standardized expression of a wide range of system hypotheses, as will be shown by translating established approaches to SYSTEMATICA 2D. This in turn will allow comparing and discussing hypotheses in a common language in order to establish common patterns, which we will refer to as 'system types', as a first step towards understanding the space of possible architectures. At the same time, the language will achieve a compromise between precise software specification and high-level description of cognitive concepts and thereby consider the other two phases, System Construction and Hypothesis Evolution.

System Construction

The difficulty of the system integration or construction process – once a system design which should satisfy all functional requirements is formulated – depends on two factors: the target infrastructure and the amount of collaboration between involved scientists. The latter does not refer to the effort of plugging together two modules in the chosen infrastructure but to the much more difficult tasks of fitting the provided output of one algorithm to the requirements of another and synchronizing the temporal order of the whole system – not to mention system-wide learning, where all modules need to support common ways of parameterization, see, e. g., [21].

Technological progress is currently mainly centered around the integration infrastructure. The most prominent examples for EI research are blackboard[19, 22, 23], service-oriented[14, 24] and data-flow-oriented[18, 20, 25] platforms. However, none of these bridge the gap between the description of a system hypothesis and the technical design of the system to be built. The resulting lack of consideration for system construction during the system formulation phases often leads to complications, usually due to underestimated dependencies between system parts built by different scientists.

We will show that SYSTEMATICA 2D combines technical and theoretical elements and adds a small but significant bias in order to enforce consideration of the construction phase during system formulation. We will show that this bias does not dramatically reduce the space of possible hypotheses but helps greatly in defining dependencies and organizing as well as speeding up the construction process. Practical applicability of the proposed language will be substantiated by presenting specific mappings of language elements to popular software infrastructures as well as experiments with large-scale intelligent systems built according to SYSTEMATICA 2D designs.

Hypothesis Evolution

After the implementation of a system hypothesis, several steps remain: the evaluation of the integrated system, the publication of the system hypothesis and the evolution, i. e. extension or revision of the hypothesis towards the next test cycle. The ease with which all of these can be done depends to a large part on the formalization of the hypothesis and the organization of the implemented system.

A modular organization may allow experiments with subsets of the full system to show the value of each system element; a standardized formalism may allow comparison to related system architectures to show novelty (see, again, [10, 5]); a modular formulation of the hypothesis will allow reuse of subsystems, possibly supported by a similar separability of the implemented system.

SYSTEMATICA 2D will provide a mathematically formalized, modular way of describing systems, combined with the ability to map this modular formulation into a modular implementation on a variety of software infrastructures, to the extent that the separability of these systems can be formally verified. The language will allow formulating technical and theoretical aspects of a hypothesis on separate levels of granularity in order to highlight relevant aspects for publication and allow theoretical discourse about system types, which we will discuss both in theory and by means of two examples of large-scale intelligent systems.

Structure of the Thesis

First, in the next chapter we will present two 'reference systems' which will serve as recurring examples throughout the thesis. The two systems are from the humanoid robotics and intelligent vehicle domains and thus are good examples of what we understand as 'Embedded Intelligence' domains. Based on these introductory examples, we move to the analysis of constraints a formalism has to fulfill, derive a measure to evaluate system formalisms and compare existing approaches in Ch. 3. We then present our formalism 'SYSTEMATICA 2D' in Ch. 4, accompanied by a discussion of the impact of guiding elements or 'structural bias'.

We continue by presenting mappings of the formalism to popular software infrastructures in Ch. 5, including generic concepts to enrich an infrastructure in order to allow a smooth translation of SYSTEMATICA 2D designs into implementations. The conceptual part of this work is completed by an analysis of the comparability of system hypotheses under SYSTEMATICA 2D in Ch. 6. Finally, the gained integration speed as well as other beneficial results of the presented concepts are shown in Ch. 7 before concluding in Ch. 8.

2 What Kind of Systems?

It is possible, but less intuitive and comprehensive, to discuss the benefits of formal design languages for abstract embedded intelligence systems. We therefore present two EI systems in this chapter which will serve as recurring examples throughout the thesis. They span two important domains of EI research, humanoid robotics and automotive safety systems, and thus represent the level of generality for which the techniques in later chapters were researched – although their applicability may be broader. Both are collaborative projects of between five and 15 scientists, both are recent scientific projects, exceeding the state-of-the-art on the algorithmic as well as on the system level and both focus on major problems of their respective domains. In other words: the presented systems are scientifically interesting and of comparable scientific and computational complexity.

For each of these systems we will describe the same relevant aspects:

- the major challenges the system addresses to overcome,
- the system hypothesis followed,
- the individual algorithms used,
- the design, implementation and evaluation process followed and
- the final system behavior and results achieved.

2.1 Adaptive Learning and Interaction System

Goal of the Adaptive Learning and Interaction System ('**ALIS**') was to enable a humanoid robot to interact freely with a human tutor and learn associations between arbitrary speech labels and generic sensor- / action-related properties of the world[1]. Such properties can be spatial (left, right, top, ...), object-related (big, small, moving, still, ...) and behavior-related (approach, grasp, release, ...)[26]. Speech labels can be arbitrary, in any language, based on a clustering of the auditory sensor space[27, 26].

2.1.1 Challenges

The major challenges towards these goals are the following: First, in order to allow real-world interaction on a humanoid platform, a generic visual and auditory segmentation of the world into background and interaction targets must be made. Second, possible behaviors of the humanoid robot must be selected such that the robot reacts to the tutor fluently, combining different elements like looking, pointing, walking and nodding, but without violating its basic constraints, e. g. pointing backwards or touching its head with its arm. Thirdly, the current sensor (position, size, motion, etc.) and behavioral (approach, retreat, grasp) state must be clustered and associated to the speech labels provided by the tutor. Finally, learned associations are to be 'expressed' by choosing appropriate actions in response to queries by the tutor. Since the implemented system is to run on a humanoid robot, relevant system parts need to run at high enough frame rates to support interactive operation, but this challenge is usually not explicitly formulated.

In addition to these functional challenges, **ALIS** was the motivating example for this work's objective to improve the system integration process. Especially the (secondary) challenges of Testability and Extendability were not sufficiently solved for an efficient development of **ALIS**.

2.1.2 System Hypothesis

A schematic view of the **ALIS** system hypothesis can be seen in Fig. 2.1. The system is split into two major parts: The lower half is composed of sensor pre-processing and action generation using a central arbiter which, together with the robot hardware and the environment, forms a reactive loop that serves as an interaction basis for all further processing. The upper half performs feature clustering (visual, auditory and behavioral), association learning, expectation generation and expectation evaluation[28].

This enables the robot to interact naturally based on the lower reactive layer, independently of the upper associative layer, unless the upper layer sends a modulation signal to activate a behavior (e. g. nodding). In addition, the associative layer can rely on a task-unspecific visual detection and tracking of interaction targets, reactive gaze tracking and following as well as modulation-friendly behavior selection.

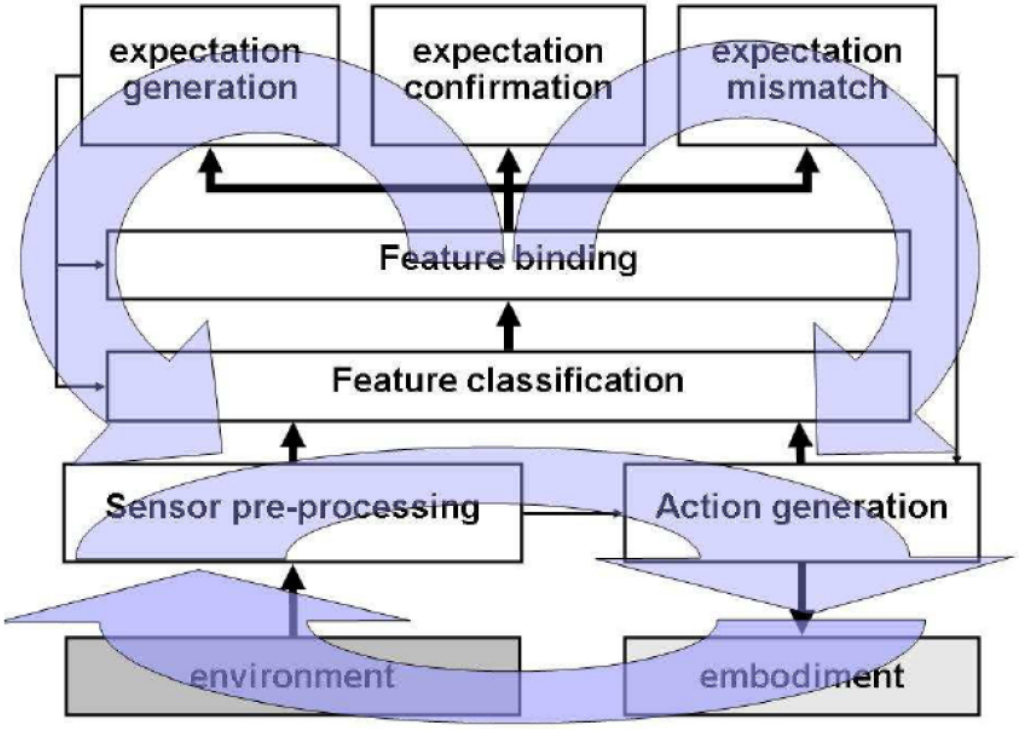

Figure 2.1: **ALIS System Schematic** – The figure shows system elements and the layered connection structure of the **ALIS** system. Taken from [1], please refer to text for details.

2.1.3 Specific Algorithms

The entire system is made possible by a large number of interacting algorithms of which we want to highlight four. All details are taken from the publications relevant to **ALIS**: [29], [30] and [1].

Proto-Object Detection and Tracking The concept of Proto-Objects stems from psychophysical modeling[31] and has been technologically applied in several works[32, 33, 29]. A Proto-Object in **ALIS** is a task-unspecific sensory detection which is stable over time, space and possibly multiple cues, e. g. a moving colored bottle which is repeatedly detected by depth-, color- and motion-segmentation. Detections within each cue are stabilized and tracked over time before they are fused among cues to stable interaction targets[29]. The data stored with each Proto-Object contains its 3D position, the 2D bounding box and pixel segmentation (for visual Proto-Object)

and a Kalman-filter for movement prediction. Given this information, most behaviors (gazing, pointing, approaching) can function based on the data of the currently active Proto-Object alone[26].

Behavior Selection During the full interaction, the robot has a variety of possible motor behaviors available: gazing or pointing at a Proto-Object, approaching a Proto-Object by walking towards it, nodding or shaking its head, grasping and releasing an object and retreating to its starting position. These behaviors must be dynamically activated and deactivated based on the current reactive preprocessing results (Are there visible Proto-Objects?), the modulation received from the associative system layer (Does the speech label 'right' match the current Proto-Object's position?) and the physical constraints of the robot (Can the robot nod and shake its head at the same time?). These problems are solved by a central 'Arbiter' in the 'Action generation' module, responsible for evaluating the current demand and feasibility of each behavior and selecting which set of behaviors should be active at any given time[29]. Behaviors compute a fitness value based on current sensor information (e. g. pointing is possible if a Proto-Object is visible) and may receive an activation bias from the associative system layer. Together with a compatibility matrix this allows reliable and flexible behavior activation.

Feature Space Clustering and Association The associative system layer works on one Proto-Object at a time. Based on the Proto-Object's data, a set of sensory features is computed, namely 3D position, size (1D) and movement speed (1D). Associations are learned between a visual sensory feature space and the space of arbitrary auditory labels within the time-window of a tutor-initiated learning session of approximately five to ten seconds with three to five repetitions of the speech label. In order to decide whether the recorded data should be used to create a new cluster in the relevant feature space or should extend an existing one, a novelty detection is employed which uses the learned associations to map cluster activations from one feature space to another. In other words: A feature cluster in position space can be extended by using the same speech label and a synonym speech label can be learned by showing the Proto-Object in the same position as a previously associated speech label[1].

Expectation Generation and Evaluation The last missing functionality of the system is to apply the learned associations to queries of the tutor. For this purpose, the system constantly creates expectations about the associated features: a speech

label 'big' from the tutor will create an expectation for a big Proto-Object, a speech label 'left' will create an expectation for a Proto-Object on the left, etc. These expectations are then evaluated on the active Proto-Object (should there be one): if the Proto-Object matches the expected features, a bias to the 'nod' behavior is generated, otherwise a bias to the 'shake' behavior, followed by a fixed timespan where the robot tries to find other Proto-Objects which satisfy the expectation[28].

2.1.4 Process

The design process followed through the two major version of the **ALIS** system[30, 1] was driven by the desire to couple specific system functionalities (e. g. Proto-Object tracking, association learning and grasping) in order to achieve specific properties of the whole system (e. g. autonomous interaction, multi-modal learning, etc.). Not all functionalities were complete when work on the system started (the first version required a close-talk microphone and did not allow grasping) and several demands on functionality performance only became apparent during system integration (e. g. association learning was extended to allow overlapping clusters). Thus, the overall system design was continuously extended as new functionalities were added or matured – which makes **ALIS**, in our opinion, a typical system integration project (cmp. Sec. 3.2.1). As a result of this interlink of design and implementation, a separation of the system was not easily possible, thus most evaluations were done on the full system.

2.1.5 Results

A full description of system behavior, evaluation results and scientific impact can be found in [1], Fig. 2.2 shows two sample scenes from an interaction with a tutor. In short, the system, running on the Honda humanoid robot, displays an unprecedented ability for autonomous interaction with arbitrary interaction partners and is able to learn generic concepts about the world (left / right), objects (big / small) and its own actions (approach, grasp) in any language. It represents a state-of-the-art intelligent artifact system, in terms of algorithms, complexity and system performance.

2.2 Automotive Multi-Cue Object Detection System

The second system which will serve as a recurring example is the Automotive Multi-Cue Object Detection System ('**AutoSys**'). It is a mainly vision-oriented system with the aim of finding and tracking all relevant traffic participants in real-world

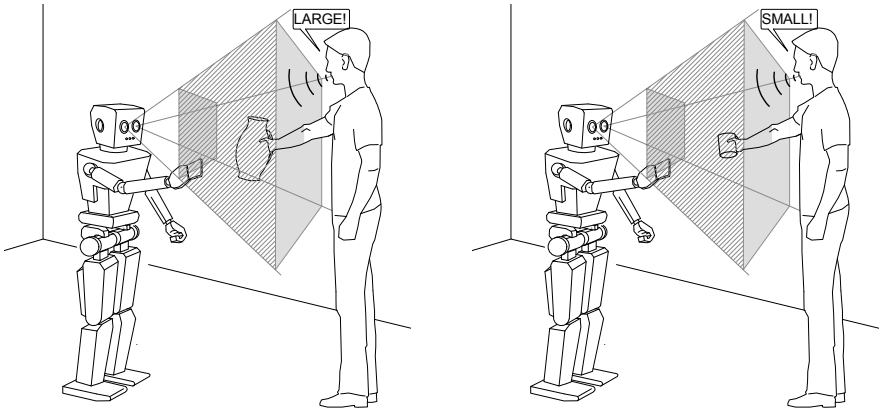

Figure 2.2: **ALIS interaction example** – The figure shows two situations from a normal interaction of a tutor with the **ALIS** system, running on the Honda humanoid robot. The robot learns the size of large (left) and small objects (right) and binds these features to words. Based on [1], please refer to text for details.

scenes under varying contexts (highway, rural road, inner city) and weather situations (sunny, overcast, rainy, snowy) both in day and night time[2]. The result is a visual warning for the driver which highlights traffic participants with a low 'time-to-contact', i. e.participants where a contact or collision with the ego-vehicle is predicted for the near future.

2.2.1 Challenges

The main challenge is the required broadness of objects and contexts: the system should not be limited to detecting objects where the visual appearance is known or to contexts where appearance-based classification works well (i. e. well-lit scenes without sharp shadows). This in turn asks for multiple, more flexible detection algorithms, together with a generic fusion process and shared representation in order to compute, e. g., the time-to-contact. This multi-cue approach however poses a new challenge since detections from more flexible detection algorithms produce a high-number of false positive (i. e. detections where no interesting object is in the world, like a window which looks similar to a car). These detections should be filtered in a way that objects in the world are found, even if only one detection algorithm found them, but still the number of false-positives is minimal.

Where **ALIS** was the motivating example, **AutoSys** served as a proof-of-concept for the system integration benefits of SYSTEMATICA 2D. We will show in Sec. 5.3

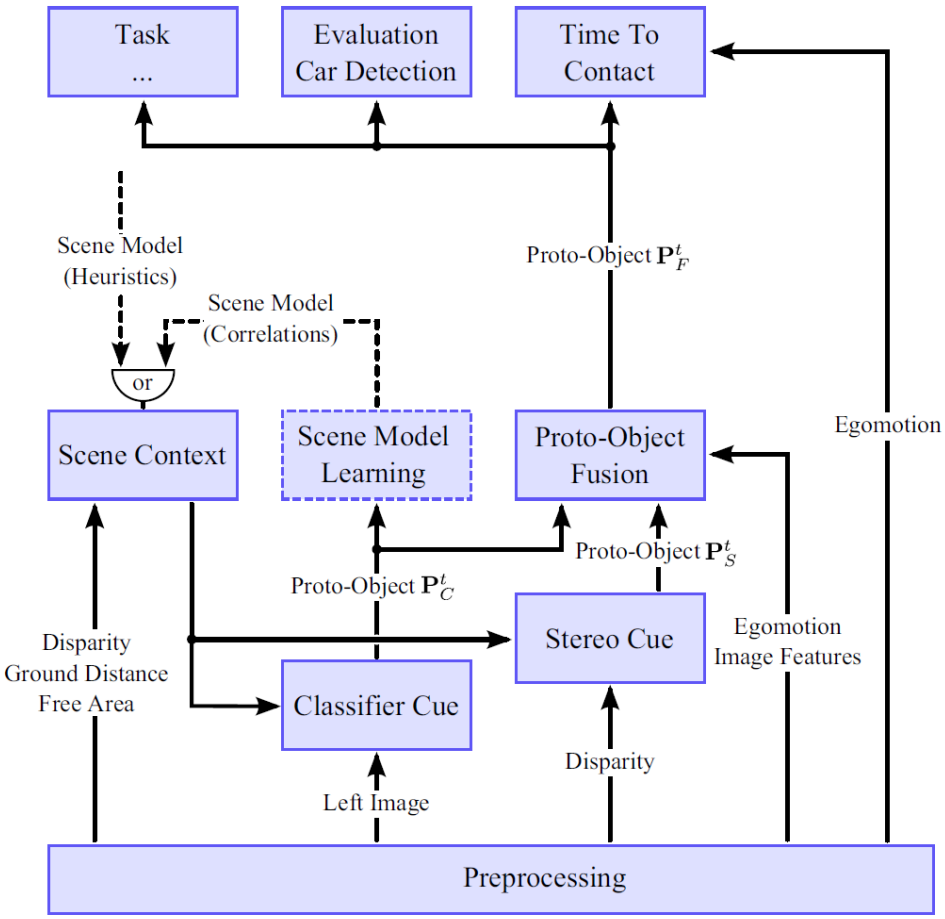

Figure 2.3: **AutoSys System Schematic** – The figure shows system elements and the layered connection structure of the **AutoSys** system. Taken from [2], please refer to text for details.

and Ch. 7 how Testability, Extendability and Reusability were improved. The fact that **AutoSys** was not built primarily to demonstrate these benefits, i. e. does not elevate them to first-class challenges, emphasizes the applicability of SYSTEMATICA 2D to real-world system integration projects.

2.2.2 System Hypothesis

A schematic of the final **AutoSys** system can be seen in Fig. 2.3. Based on a common preprocessing performing disparity calculation, ground plane estimation, road surface detection and ego-motion estimation, two detection cues produce a set

of Proto-Objects in each camera frame (for an introduction to Proto-Objects see Sec. 2.1.3). This detection is modulated by a 'scene context' model, either learnt based on detection statistics or in form of a heuristics optimized offline, which helps to filter out unreasonable detections (e. g. cars above the ground plane, too big or too small, wrong aspect ratio, etc.). The filtered detections are fused and stabilized over time, space and other image features, again reducing spurious detections, before being used by task-specific post-processing algorithms like the time-to-contact computation[2].

2.2.3 Specific Algorithms

Like with the **ALIS** system, a large number of algorithms contribute to the performance of the overall system. We will not focus on the (equally sophisticated) preprocessing, including disparity calculation[2], road surface detection[34] and ground plane estimation[35], and rather give a short summary of the following four 'highlights'. All details are taken from the publications relevant to **AutoSys**: [35] and [2].

Detection Cues Two algorithms are used as visual detection cues, one appearance-based and thus task-specific and one disparity-based and thus task-unspecific. The appearance-based detection is based on a visual classification of patches in an image pyramid of different resolutions[36]. As a result, the same object model can be used to detect objects of different sizes resulting in activations at different levels of the pyramid. In order to estimate the regions of interest (ROIs) of detected objects, a single layer perception is trained on the activations at different scales, leading to detections with object identity, ROI and classification confidence[2].
The disparity-based detection uses a region-growing on the disparity map, resulting in detections with ROI, 3D position, physical size and distance. For detections from the appearance-based cue, 3D position, physical size and distance are computed based on the visual ROI and the disparity map.

Scene Context Modulation In order to provide a first filter to reduce the large number of spurious or even stable incorrect detections from both cues, results are filtered with a 'scene context model'[2]. This model is essentially a set of constraints on the relevant characteristics of the detections: 2D position, ROI aspect ratio, height above ground plane (based on 3D position and ground plane) and physical size. Two approaches were tested: Using a multi-objective optimization based on the

method in [37], a set of bounds for each characteristic feature can be found, leading to a binary decision heuristics. Alternatively, feature distributions associated with object identities were learned based on the method in [21] and then inverted in order to obtain probabilistic distributions over the features and thus modulate detection confidences to achieve soft filtering[2].
As a result, the number of false-positives (i. e. detections where no interesting object exists) can be drastically reduced, almost without dropping correct detections.

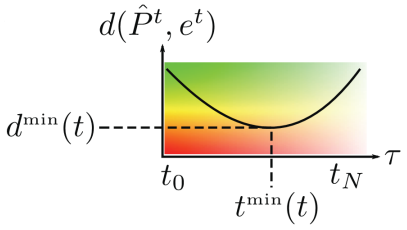

Figure 2.4: **Color coding of time-to-contact estimations** – from [2].

Advanced Proto-Object Fusion Based on the Proto-Object treatment used in **ALIS** (see Sec. 2.1.3 or [26]) the fusion process was extended to deal with the high number of possible detections in a realistic, cluttered out-door environment. Main difference is the inclusion of alternative features into the fusion process: based on the Proto-Object ROI, color and Local Orientation Coding (LOC)[38] features are extracted from the original image and stored with the Proto-Object. When fusing new detections with the Kalman-predicted previous set of Proto-Objects, a comparison of these features in addition to the position and ROI give much less support to spurious detections and thus again reduce the number of false-positives[2].

Time-To-Contact Estimation As an example of the Proto-Object based postprocessing, the time-to-contact estimation uses the driver's own predicted trajectory at time $\boldsymbol{t}$ ($\boldsymbol{e^t}$, based on speed and steering angle) together with the predicted trajectories of Proto-Objects at time $\boldsymbol{t}$ ($\boldsymbol{\hat{P}^t}$, as encoded in their Kalman-filter) to estimate the relative distance $\boldsymbol{d(\hat{P}^t, e^t)}$ over future timesteps $\boldsymbol{\tau}$. The minimal distance $\boldsymbol{d^{\min}(t)}$ and associated time $\boldsymbol{t^{\min}(t)}$ are then computed and shown to the driver using a simple color coding (see Fig. 2.4).

2.2.4 Process

Similar to the **ALIS** system, most specific algorithms incorporated in **AutoSys** were not finalized when the integration process started. However, the system hypothesis, and with it the central question to be answered by the full system, was formalized at the beginning: "Can several noisy detection cues be combined with context filtering

(a) Stereo Cue (b) Classifier Cue

(c) Fused (d) Time To Contact

Figure 2.5: **AutoSys visualization example** – The figure shows the visualization of different stages of computation in the **AutoSys** system. Taken from [2], please refer to text for details.

and Proto-Object fusion so that all relevant objects are found under many different real-world driving conditions?" As a result, the overall system design, as well as most dependencies between algorithms, was static throughout the implementation process. Experiments and evaluations could be done on subsystems, e. g. with only a single detection cue or with disabled scene context, in order to judge the contribution of each part to the performance of the whole system.

2.2.5 Results

A full description of system behavior, evaluation results and scientific impact can be found in [2]. Fig. 2.5 shows a visualization of processing results at different stages of computation. As a first step, detections of the appearance- and disparity-based cues are shown (after scene context filtering). These are then fused (third image) and used for time-to-contact estimation (fourth image, color code see Fig. 2.4). System performance is shown to increase substantially because of scene context filtering and

Proto-Object fusion, in direct sunlight as well as in overcast, rainy, snowy situations and at night. In short, the system is shown to surpass existing visual detection system in the automotive domain, making it a state-of-the-art intelligent artifact system.

2.3 Conclusion

This chapter has introduced two implemented system instances, their underlying hypotheses and the major system elements. Both systems work under real-world conditions, combine methods from various disciplines (e. g. image processing, tracking, supervised and un-supervised learning and robot control), were built by groups of between five and 15 scientists and represent the state-of-the-art in their respective fields.

On the one hand, these systems will serve as recurring examples in all future chapters. They represent two examples of the 'Embedded Intelligence' domain in that they aim to provide intelligent capabilities to physical, real-world artifacts. They were both designed and built in a research environment and thus allow to discuss the specific requirements of system design formalisms for this domain (see Ch. 3). In addition, since both systems were built based on some form of formalism (**ALIS** using an ad-hoc notation as shown in Fig. 2.1, **AutoSys** using the SYSTEMATICA 2D language introduced in this work), they will serve as examples for these two languages and make their comparison more intuitive.

On the other hand, we can derive generalized goals and motivations from the problems encountered and overcome in the process of designing and constructing both systems. These are generally rooted in the research process, which is based on an iterative, spiral refinement of concepts, system elements and full systems, with a strong collaborative component in all three phases.

The conflict between each scientists individual work on the algorithms in components or system units and the collaboration between these units in the full system is the most dominant obstacle in the system construction process. First, the collaboration in a system with unclear dependencies is slowed down by changes in different system parts interfering with each other, thereby dramatically increasing the time needed for system integration. Second, unforeseen dependencies between system parts require incremental change of the original design or system hypothesis during the system construction phase.

While the first issue entails merely a waste of resources, the second can seriously affect the quality of scientific statements about the original system hypotheses: If the

implemented system differs in design from the original hypothesis, what conclusions about this hypothesis do the experiments on the implemented system allow?

We believe that one method to overcome these limitations is a way to describe systems which is both able to capture the theoretical aspects of the system hypothesis and to guide and support the system construction. The remainder of this work will therefore be concerned with motivating, introducing, evaluating and discussing such a new formalism for system design of intelligent systems. While we will demonstrate the main benefits of this new formalism on the contrast between **AutoSys** (built with SYSTEMATICA 2D) and **ALIS** (built without), the following inductive steps are targeted at the generalized set of problems, not at the specific system instances.

3 A Measure for Design Languages

As a first step to improving the formalization of EI system hypotheses, this chapter will address the questions how one formalism can be compared to another and what strengths and weaknesses existing formalisms have. To obtain a good and complete measure for system formalisms, all three phases described in Ch. 1 must be taken into account: Hypothesis Formulation, System Construction and Hypothesis Evolution. We will refer to the iteration over these three phases as the '*Hypothesis Test Cycle*' (see Fig. 3.1) and define a set of criteria to form the measure in such a way that they evaluate the suitability of formal notations over the whole cycle. This includes the flexibility and comparability of descriptions, addressing of future implementation issues and the ability to decompose both the design and the implemented system.

Since the comparison of system formalisms is a central part of this thesis, related work will be discussed in three parts. Sec. 3.1 will cover the wider area of EI system notations and taxonomies for such notations. Sec. 3.3 will introduce a set of criteria, following a discussion of the specific requirements of EI system integration: the relation to software engineering practices (e. g. UML[16]) and to software infrastructures (e. g. CAST[39], XCF[22], YARP[20]) as well as the importance of guiding the design process by a 'structural bias' in the formalism. In Sec. 3.4 several existing formalisms are then evaluated along those criteria: the class of ad-hoc notations used in only one publication (we will refer to these as 'Boxes and Arrows'), 3-Tier[3], CogAff[4] and SYSTEMATICA[5]. A conclusion summarizes the lessons learnt from defining and applying the measure criteria to existing formalisms. Major parts of this chapter were previously published in [40].

3.1 Related Work

Two areas of related work are relevant to this work: measures of formal notations ('taxonomies') 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

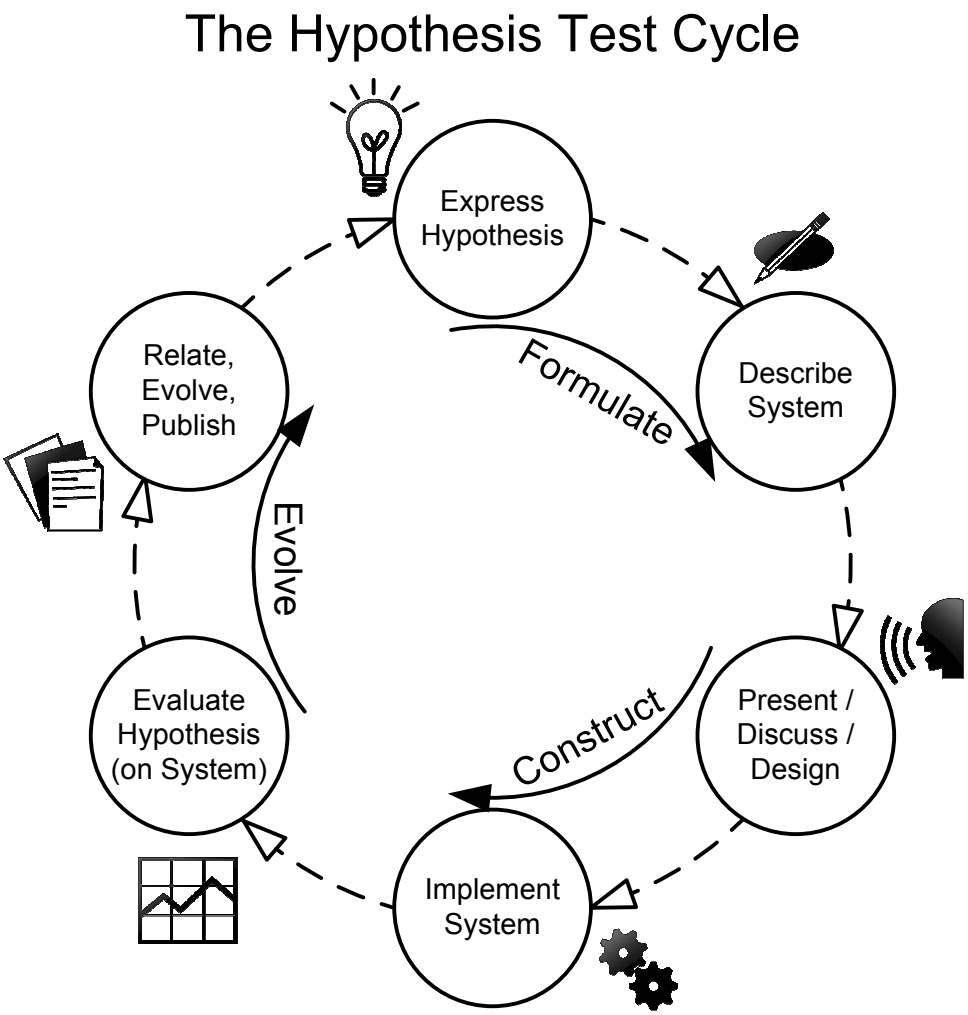

Figure 3.1: Visualization of the Hypothesis Test Cycle.

the work of Medvidovic et al.[41] giving a qualitative comparison of software architecture description languages (ADLs). 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[42], C2, LILEANNA and ACME[43], and determine which of them can be considered an ADL. One of the conclusions we can strongly agree with 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.[8] 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.[5], where a new framework for modeling hierarchical architectures ('SYSTEMATICA', encapsulating the subsumption architecture[11]) is used to express existing cognitive architecture approaches 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 (in contrast to the elements of the evaluated systems) affect their cognitive qualities.

Formal notations for intelligent systems today come from three areas. First, there are mathematical formalizations of system component interaction[44, 45, 46]. 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[3] or CogAff[10]. Second, architecture description languages are a popular tool in the software engineering domain to describe large software systems, for instance Rapide[42] and ACME[43], 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 guidance, or 'bias', they can provide for guiding an EI system design in a favorable direction (see discussion of structural bias in Sec. 3.2.2). Especially xADL provides no structural bias whatsoever by itself but rather allows expressing architecture constraints. It is therefore reasonable to assume that the formalism introduced in this thesis can be expressed using xADL, but this is no limitation of the concepts put forward here.

Finally, there are specific notations used in intelligent systems[11, 4, 19, 9, 12, 47, 48] or reviews[3] and plans[49] 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[3], used in [12], CogAff[10], used in [4], and SYSTEMATICA[5], used in [9]. We will perform a more detailed analysis of these four notations, 'Boxes and Arrows', 3-Tier, CogAff

and SYSTEMATICA— the focus being on the comparison of the notations, not of the systems built with them — when we evaluate them in Sec. 3.4.

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 the next chapter.

3.2 Target Setting

To formulate a specific set of criteria for a formal language, three considerations are central: 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.

3.2.1 Why EI System Integration Is Not Software Engineering

What are the scientific challenges in EI system integration? 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[50] 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. It is this combination of a high degree of spread expertise among scientists and the co-evolution of system and components which we see as typical traits of software integration projects in the intelligent artifact domain.

These differences lead us to the conclusion that the perfect formalism cannot be the most exact one. Languages like UML[16] 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 [51]). In other words, while software engineering is focused on creating designs in the sense of specifications, system design for research and collaboration needs a stronger foundation in concepts.

3.2.2 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'. The degree of structural bias can be estimated by counting the number of constraints which have to be considered while designing a system: how many system elements are pre-defined, which combinations / connections are not allowed, etc. Fig. 3.2 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

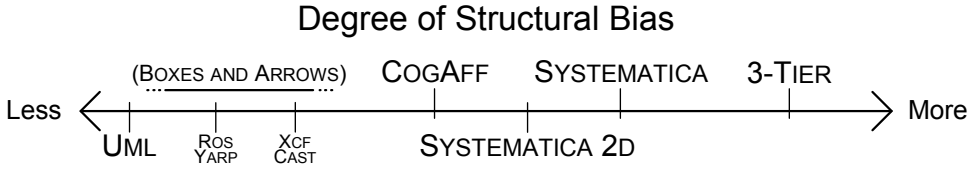

Figure 3.2: Rough ordering of formalization approaches according to the amount of structural bias.

in Secs. 3.4 according to the criteria defined in the following; the new formalism Systematica 2D introduced in this thesis will be evaluated in Sec. 4.3.2.

3.2.3 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 **C**oSy **A**rchitecture **S**chema **T**oolkit aims at "construction and exploration of information-processing architectures for intelligent systems"[39]. A 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 **X**ML enabled **C**ommunication **F**ramework[19, 22] 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[52], YARP[20], ROS[18], ToolBOS[25] and OpenRTM[53]. They all share the basic decomposition of a system into concurrently running processing components and support their configuration, communication and monitoring. All except ThinkingCap and ToolBOS support the addition and removal of components at run-time, the 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 [52] 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, Ch. 5 will discuss the mapping of the SYSTEMATICA 2D formal design language introduced in this thesis to a set of software infrastructures.

3.3 Criteria for System Integration Formalisms

Only few attempts at establishing a formalism measure have been made (e. g. [41, 8]) and even those are qualitative in nature. We agree to the assessment made in these works: A measure to judge the suitability of a descriptive language to qualitative demands, most of them subjective, will be qualitative itself, asking the right questions about a formalism, but without the means to judge their fulfillment quantitatively. We will therefore use the measure proposed in the following with caution and revisit the lessons we can learn from judging a formalism with this measure when we do so in Sec. 3.4.

That being said, we will now formulate criteria relating to the three main considerations of the hypothesis test cycle: criteria A1-A3 specify the required minimum expressiveness for system design; criteria B1-B3 specify additional structural bias for collaboration; criteria C1-C3 add structural bias for efficient implementation.

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.

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 and detailed enough to aid construction of the system on the other hand.

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.

Description of Interfaces

B1: A formalism should allow the specification of the interfaces of system elements (units) whenever they affect at least two collaborating scientists — following the arguments in Sec. 3.2.1, the main purpose of the design can only be to describe what is between individual scientist's fields of work, most notably their interfaces.

Decomposition to Individuals

B2: A formalism should allow a 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).

Description of Dependencies

B3: A formalism should allow specifying the dependencies between collaborating scientists and identify tightly or loosely coupled interaction — along the same lines as B2, each individual working on a unit should be at least aware of the units required for his or her work.

Translation to Infrastructure

C1: 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.

Exploitation of Infrastructure

C2: 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 (B2) and express their dependencies (B3), 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.

Subsystem Separation

C3: A formalism should allow separating an existing system into subsystems and reusing or extending its subsystems by means of the decomposition in the implementation (C1) and the formal description of dependencies (B3) — reuse of single units is good, exploiting decomposition and dependencies to allow reuse of larger building blocks is better.

3.4 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 numerous publications), 3-Tier (as a popular example of technical embedded system modeling, see [3, 12]), CogAff (as an example of a biologically motivated design, see [4]) and SYSTEMATICA (as a formal language for analysis of hierarchical architectures, see [5]). 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 Sec. 3.3, 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?

3.4.1 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 Sec. 3.3 we can formulate the strong and weak points of this kind of ad-hoc formalization (see Fig. 2.1 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.

B,C (–) 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.

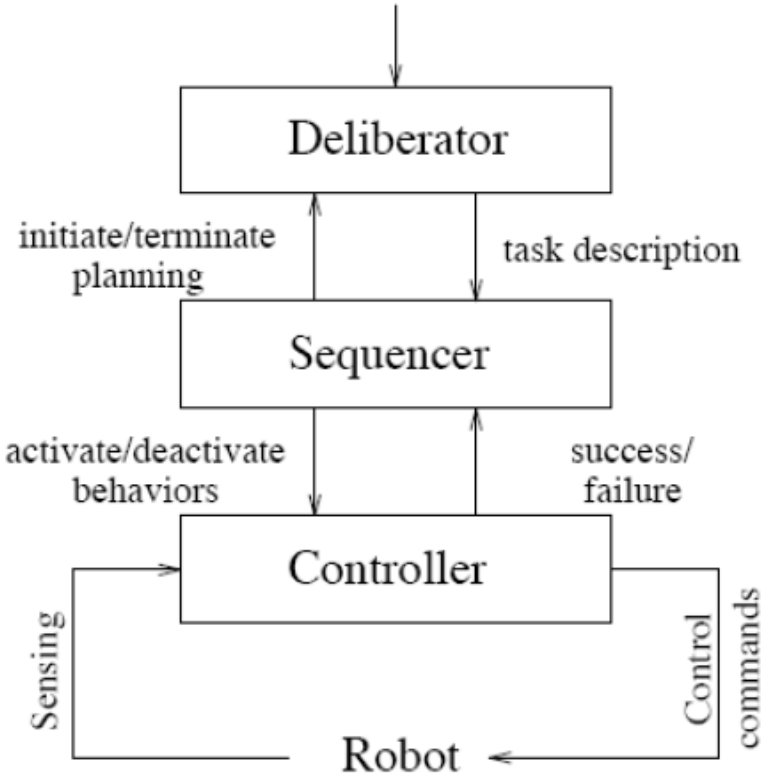

Figure 3.3: Schematic view of the 3-Tier architecture skeleton as proposed by [3].

3.4.2 3-Tier Architectures

The class of 3-Tier architectures, as presented by Gat[3] and used more recently e. g. in [12], is mainly used for robot control where reactive and deliberative systems work together (see Fig. 3.3).

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.

B1 (?) The language does not directly formalize the data transmitted between layers, but the nature of all communications is implied by the concept.

B2 (–) The rough, three-part decomposition cannot express the separation of individual scientist's work packages.

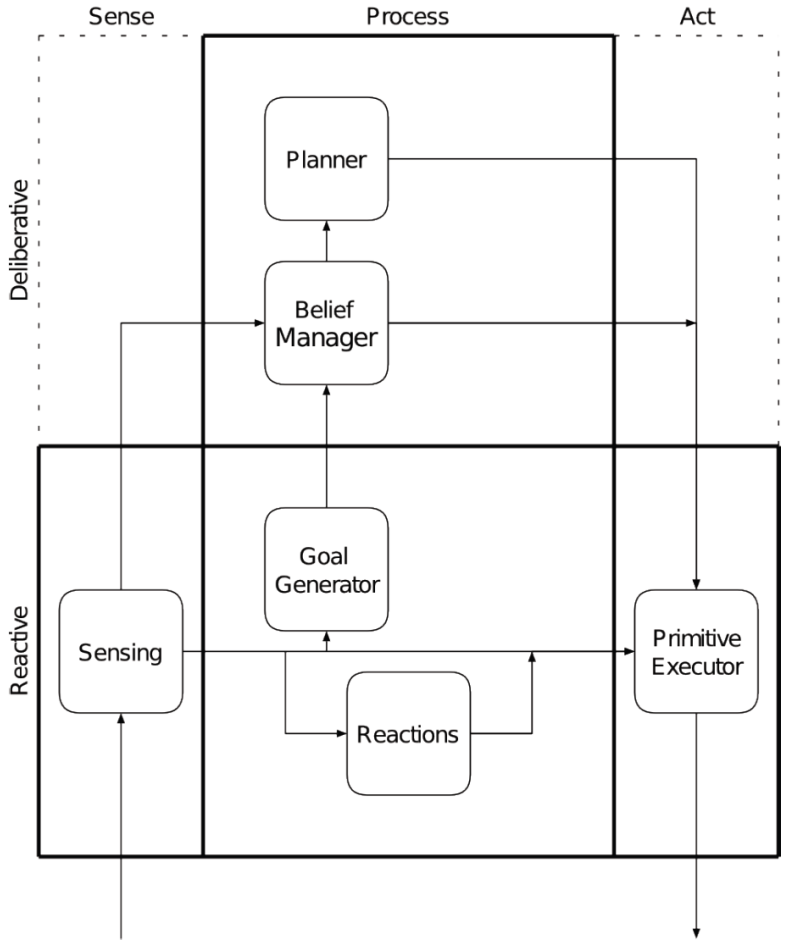

Figure 3.4: Example of an agent design in the CogAff framework[4].

B3 (+) Based on the original concept, a tight coupling from bottom to top and a loose coupling from top to bottom are implied.

C1,2 (+) 3-Tier is traditionally used for robotic applications, therefore there are many examples of implementations.

C3 (?) Although the design units can be translated to infrastructure units, their rough granularity makes it unlikely that they can be reused without modification in a subsequent implementation.

3.4.3 CogAff

The CogAff architecture schema presented by Sloman et al.[10] 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[4] to evaluate the power of this formalism, see Fig. 3.4.

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[10] nor in the sample application[4].

B1 (–) A specification of interfaces is not included.

B2,3 (+) Fine decomposition, focusing on single scientists is possible, dependencies are not specified but can be derived from the positioning.

C1,2 (+) Although no specific infrastructure is mentioned in [4], a mapping of the units and connections to a standard middleware like YARP or ROS seems straightforward.

C3 (?) Design units translate to implementation units, missing interfaces and implied dependencies make reuse of subsystems unpredictable.

3.4.4 Systematica

The Systematica formalism introduced by Goerick[5] aims at providing a uniform description language for hierarchical system architectures. It takes the idea of incremental system layers (as also found in Subsumption[11]) and adds bottom-up representation and top-down modulation channels, see Fig. 3.5.

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.

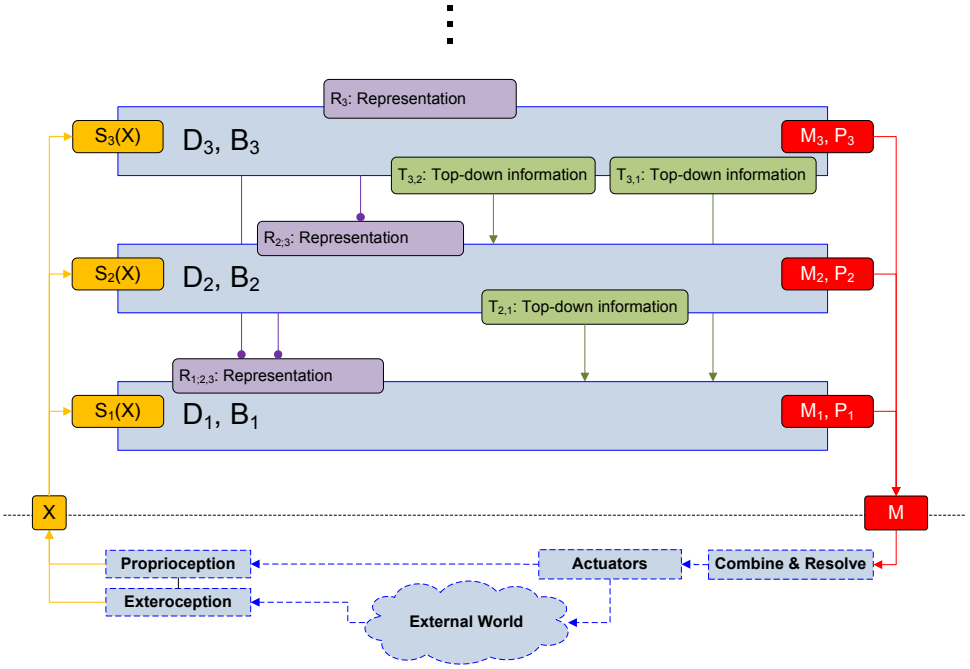

Figure 3.5: Schematic view of the SYSTEMATICA formalism. Taken from [5].

B1 (+) Interfaces are specified by representation and top-down output; input ports are implied.

B2 (–) The constraint of full sensor-motor loops is often too rough for a decomposition to individual scientists.

B3 (+) Coupling and dependencies are specified by the top-down and bottom-up channels.

C1,2 (+) A translation to the ToolBOS[25] infrastructure is presented in [9] and allows partial execution along the bottom-up / top-down dependencies[54].

C3 (?) Design units translate to implementation units, rough decomposition makes reuse without modification difficult.

3.5 Conclusion

This chapter has motivated, introduced and applied a set of criteria to evaluate the suitability of a system notation for the Hypothesis Test Cycle. Major points of the motivation were the difference between EI research and software engineering, the role of structural bias and the descriptive powers of popular software infrastructures. The formulated criteria focus both on the descriptive abilities of the evaluated formalism

(general, standardized, meaningful, interfaces) as well as on the implementation-orientation (decomposition to individuals, dependencies, mapping to / exploitation of infrastructure, subsystem reuse). These criteria are qualitative, their application is therefore a matter of argument, but even so they allow identifying the strong and weak points of a given notation. Our basic question when evaluating a notation according to each criterion was whether the notation explicitly enforces the specification of the requested piece of information.

According to the introduced criteria, four popular EI system notations were evaluated: ad-hoc 'Boxes and Arrows', 3-Tier, CogAff and SYSTEMATICA. This evaluation 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 Sec. 3.2.3, software infrastructures are not able to design the functional aspects of these systems, like communication patterns, incremental construction, 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 Sec. 3.2.1).

We believe that the formulated criteria allow one 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 (fast construction) as well as define interfaces, dependencies and decomposition to individuals (research flexibility). Based on this understanding of benefits and drawbacks, the next chapter will introduce the SYSTEMATICA 2D formalism for system design, which aims at combining a flexible system description with a structural bias for system design and easy extendability in order to satisfy the introduced criteria in all phases of the hypothesis test cycle.

4 A New Design Language

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. Arbitrary '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 description of interfaces or semantics; SYSTEMATICA in turn supports interfaces and semantic description but is 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 the new EI system design language 'SYSTEMATICA 2D' (short: 'SYS2D'). The language is designed to support the hypothesis test cycle along all three phases: to provide a flexible and standardized description, allow a fast and uncomplicated implementation and improve decomposition and reusability of both design and integrated system.

In this chapter we introduce the language with its formal and associated visual notation and then discuss several related issues. Most notably, the language introduces a small but effective structural bias which allows to derive and prove several properties of systems built according to a SYSTEMATICA 2D design, e. g. the ability for partial testing and global deadlock-free operation. The same structural bias changes the way some design patterns are realized and we will discuss this impact on the example of standard patterns (e. g. server-client and the 'lateral support' pattern).

To evaluate the new language we will show that it can express the three related formalisms evaluated in Ch. 3 as well as relate it to the same criteria. In addition, we will present the **AutoSys** design in SYSTEMATICA 2D and discuss how this affected the implementation process (since **ALIS** was not built based on a SYSTEMATICA 2D design, the comparison of both systems will follow in Ch. 7). Major parts of this chapter were previously published in [40].

4.1 Systematica 2d Language Specification

A Systematica 2d system is a combination of descriptions on two levels: the functional and the descriptive. This is beneficial for two reasons: First, in order to provide a language which is both suitable for EI system hypothesis description and allows efficient implementation, it is better to separate the description (especially the granularity) of functional and semantic system description. Second, the structural bias can be imposed only on the functional description in order to remove design patterns with negative impact while reducing design flexibility as little as possible.

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 sensor and behavior spaces. Fig. 4.1 shows an example design, all relevant elements will be detailed in the following.

4.1.1 Functional System Design

On the functional level, a Sys2d system is composed of the set $\boldsymbol{U}$ with $\boldsymbol{N > 2}$ processing units $\boldsymbol{u_n \in U, n = 1..N}$. There is always one unit $\boldsymbol{u_1}$ representing and emitting sensor events from exteroception $\boldsymbol{S_e}$ and proprioception $\boldsymbol{S_p}$ which together form the full sensor space $\boldsymbol{S = S_e \times S_p}$. A second predefined unit $\boldsymbol{u_2}$ represents receiving and executing motor commands from the motor space $\boldsymbol{M}$. $\boldsymbol{S}$, $\boldsymbol{S_e}$, $\boldsymbol{S_p}$ and $\boldsymbol{M}$ are vector spaces.

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. 4.1):

- 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, data type and input role (defined in the following), 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 data 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 below;

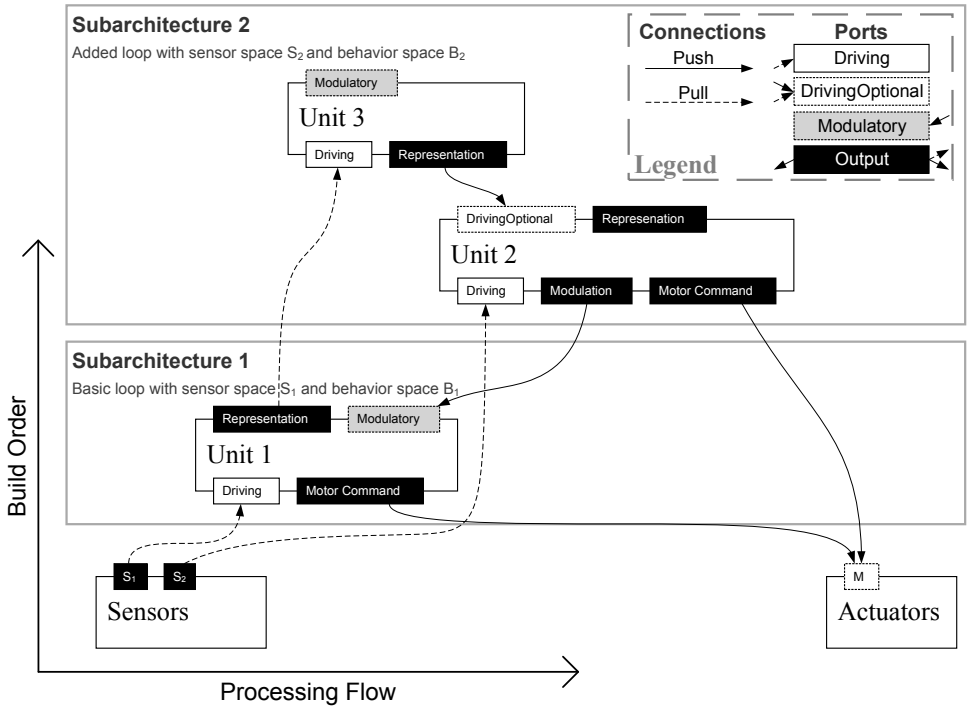

Figure 4.1: **Visualization of a** SYSTEMATICA 2D **description** – The system is composed of units which are arranged along the two axes according to processing flow and build order. Black ports are outputs, light gray ports are Modulatory inputs, white ports with solid line are Driving inputs and white ports with dashed lines are DrivingOptional inputs.

- 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 $\mathbf{Pull}_n \subset \{(u_{\text{source}}, \boldsymbol{o'}, \boldsymbol{i})\}$;
- 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 $\mathbf{Push}_n \subset \{(u_{\text{target}}, \boldsymbol{i'}, \boldsymbol{o})\}$.

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, subsystem 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 Sec. 4.2.1.

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 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 Bottom-Up and Top-Down channels in SYSTEMATICA to the Controller-Sequencer-Deliberator sorting in 3-Tier — not to mention the 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. 4.2):

- **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.

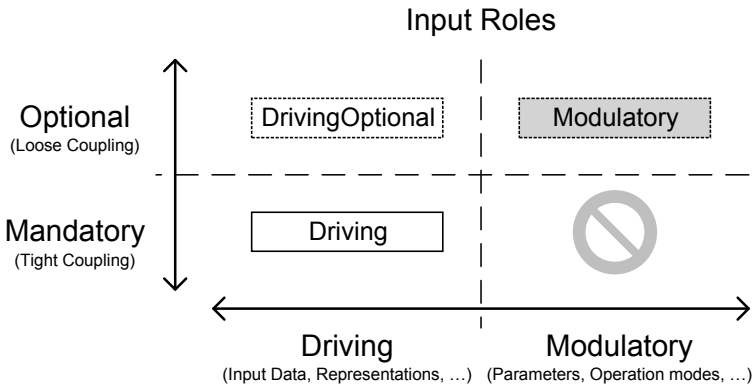

Figure 4.2: **Input roles in** SYSTEMATICA 2D – Two criteria are interleaved: driving / modulatory inputs (sometimes referred to as bottom-up / top-down) and mandatory / optional inputs. Three of these combinations are supported by the designated roles, the combination 'mandatory and modulatory' is excluded by design. See text for details.

- **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 C3: 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. Specific examples of the impact of this constraint will be discussed in Sec. 4.2.2.

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 $\boldsymbol{u_n}$.

4.1.2 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 definition of 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 distinction 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 distinction of push and pull connections allows subsystem separation in a much stronger way than by the unit dependencies alone, namely into incremental construction blocks (definition follows in Sec. 4.2.1) 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 Ch. 6.

4.1.3 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 [5]: The sensor space of a unit or set of units describes which subset of sensory signals (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 subsystems have access to specific (e. g. visual) sensory signals and which subsystems 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 the 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 $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$, a sub-architecture $\boldsymbol{a_k} \in \boldsymbol{A}$ is a tuple $\boldsymbol{a_k} = (\mathbf{name}, \boldsymbol{U_k}, \boldsymbol{S_k}, \boldsymbol{B_k})$ with $\boldsymbol{U_k} \subset \boldsymbol{U}$ and $\forall(\boldsymbol{k}, \boldsymbol{l}) : \boldsymbol{U_k} \cap \boldsymbol{U_l} = \emptyset$ (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. 4.1 for a complete SYS2D design).

4.1.4 Visual Representation

To allow faster understanding and communication, a visual representation of the SYS2D description, so far described in set notation, is clearly preferable. Fig. 4.1 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. Connections are shown as dashed lines for pulls and as solid lines for pushs. Thus, the visual representation can cover all properties of SYS2D systems. Based on this visual representation, a visual editing software was created, please see Appendix 1.

4.2 Structural Bias

4.2.1 Definition and Proof of System Properties

After introducing SYSTEMATICA 2D in its general form we will now formulate constraints on the functional side of a SYS2D design (specifically, the set $\boldsymbol{U}$) in order to provide a structural bias towards the measure criteria A1-C3. The four system properties discussed in the following are:

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.

In the following we will describe the definition of each of these properties and then derive the constraints it imposes on $\boldsymbol{U}$.

Sortability means that the units $\boldsymbol{u_i} \in \boldsymbol{U}$ can be ordered, or sorted, along the two dimensions of SYSTEMATICA 2D, which are defined by two sorting relations, $<_{\boldsymbol{v}}$ for vertical sorting along the build order and $<_{\boldsymbol{h}}$ for horizontal sorting along the processing flow.

Definition 4.1 (Sorting Relations) *A unit $\boldsymbol{u_n} \in \boldsymbol{U}$ will be called **'below' $\boldsymbol{u_m} \in \boldsymbol{U}$** (or: **v**ertically smaller $\boldsymbol{u_n} <_{\boldsymbol{v}} \boldsymbol{u_m}$) iff there is a push $\boldsymbol{p} \in \mathbf{Push}_{\boldsymbol{m}}$ connecting an output of $\boldsymbol{u_m}$ to an input of $\boldsymbol{u_n}$ or there is a pull $\boldsymbol{p'} \in \mathbf{Pull}_{\boldsymbol{m}}$ connecting an input of $\boldsymbol{u_m}$ to an output of $\boldsymbol{u_n}$. A unit $\boldsymbol{u_n} \in \boldsymbol{U}$ will be called **'left of' $\boldsymbol{u_m} \in \boldsymbol{U}$** (or: **h**orizontally smaller $\boldsymbol{u_n} <_{\boldsymbol{h}} \boldsymbol{u_m}$) 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 $\boldsymbol{u_n}$. To detect loops in these relations, we define the transitive hulls $<_{\boldsymbol{h}}^{+}$ 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* $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$ *and adding* $(\boldsymbol{a}, \boldsymbol{c})$ *to* $<^+$*.*

Definition 4.2 (Sortability) *A system* $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$ *will be called sortable iff for every relation* $<^* \in \{<_v, <_h\}$ *it is possible to assign order numbers* $\boldsymbol{o_i} \in \mathbb{N}$ *to every unit* $\boldsymbol{u_i} \in \boldsymbol{U}$*,* $\boldsymbol{i} = \boldsymbol{1..N}$*, such that*

$$\forall (\boldsymbol{i}, \boldsymbol{j}) : \boldsymbol{u_i} <^* \boldsymbol{u_j} \Rightarrow \boldsymbol{o_i} < \boldsymbol{o_j}$$

where $<$ *denotes the 'smaller than'-relation on natural numbers.*

Proposition 4.1 (Sortability of a SYS2D System) *A system* $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$ *is* ***sortable*** *iff the relations* $<_h^+$ *and* $<_v^+$ *are antisymmetric, i. e. if there is no pair* $(\boldsymbol{a}, \boldsymbol{b})$ *with* $\boldsymbol{a} <^+ \boldsymbol{b}$ *and* $\boldsymbol{b} <^+ \boldsymbol{a}$ *in either relation.*

In other words, 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 Sec. 4.2.2).

Proof 4.1 *Since the sorting relations are defined independently, sorting in horizontal and vertical direction can also be done independently and equivalently with* $<^+$ *representing* $<_h^+$ *and* $<_v^+$*. The weakest form of sorting or assigning order numbers is provided by partially ordered sets[55], 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 system* $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$ *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[55].

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 4.3 (Incremental Construction) *A system* $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$ *will be said to* ***allow incremental construction*** *if it is sortable and for every unit* $\boldsymbol{u_n}$*, for every Driving input port* $(\boldsymbol{i}, \boldsymbol{t}, \boldsymbol{Driving}) \in \boldsymbol{I_n}$ *there is a pull connection* $(\boldsymbol{u_m}, \boldsymbol{o}, \boldsymbol{i}) \in \mathbf{Pull}_{\boldsymbol{n}}$ *providing data to that port.*

This property is formulated purely in the form of a constraint on the handling of Driving inputs in order to achieve the third property, **Completeness of subgraphs**:

Definition 4.4 (Executable Subgraph) *In a system* $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$*, a subgraph* $\boldsymbol{U_k} \subset \boldsymbol{U}$ *will be called executable if all mandatory inputs of all units* $\boldsymbol{u_n} \in \boldsymbol{U_k}$ *receive data from within the subgraph.*

Proposition 4.2 (Complete Subgraph) *In a system* $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$ *which allows incremental construction, a subgraph* $\boldsymbol{U_k} \subset \boldsymbol{U}$ *is executable if it contains the full transitive subgraph under* $<_{\boldsymbol{v}}$ *of each contained unit:*

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

Proof 4.2 *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* $<_{\boldsymbol{v}}$*. By transitive completion from* $<_{\boldsymbol{v}}$ *to* $<_{\boldsymbol{v}}^{+}$*, each unit is related to all units required to provide all Driving and therefore all mandatory inputs.*

■

In other words, by constraining the use of Driving inputs to pulling connections, we can ensure that subgraphs are incrementally complete under $<_{\boldsymbol{v}}^{+}$, and thus also allow 'incremental construction' (they can be built and tested incrementally) and 'graceful degradation' (they can function even if higher units fail).

The final property achieved by the constraints of sortability and incremental construction is **Global deadlock-free operation**:

Definition 4.5 (Global deadlock-free operation) *A system* $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$ *will be said to allow global deadlock-free operation iff it is composed of locally deadlock-free units (i. e. given all mandatory inputs, all units will always produce outputs in finite time) and contains no loops of mandatory connections.*

The reason for this way of defining global deadlock-free operation is that in a system of locally deadlock-free, asynchronously running units, a deadlock on the system level can only occur by a set of units waiting on each other, which implies a loop of mandatory connections.

Proposition 4.3 (Global deadlock-free operation) *Every system* $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$ *which is composed of local deadlock-free units and is sortable is globally deadlock-free.*

Proof 4.3 *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* $\boldsymbol{u_n} <_{\boldsymbol{h}} \boldsymbol{u_m}$*. Since the system is to be sortable, the transitive hull* $<_{\boldsymbol{h}}^{+}$ *of* $<_{\boldsymbol{h}}$ *is required to be antisymmetric, i. e. there is no pair* $(\boldsymbol{u_n}, \boldsymbol{u_m})$ *with* $\boldsymbol{u_n} <_{\boldsymbol{h}}^{+} \boldsymbol{u_m}$ *and* $\boldsymbol{u_m} <_{\boldsymbol{h}}^{+} \boldsymbol{u_n}$*. Therefore, the system cannot contain loops of mandatory inputs.*

■

Definition 4.6 (Valid SYS2D Design) *A* SYS2D *system* $\mathbb{S} = (\boldsymbol{U}, \boldsymbol{A})$ *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, complete subgraphs 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.

4.2.2 Impact of Structural Bias

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. By analyzing different communication and system patterns we will show in the following sections and in Ch. 6 that these constraints do not reduce the space of possible designs in practice, even though they sometimes require a specific way of formulating them. In this section we will present two technical communication patterns (Client-Server and Publisher-Subscriber) and one popular artificial intelligence design pattern, 'Lateral Support',

to illustrate the SYS2D constraints on real-world examples and show how they influence the formulation of these designs.

Server-Client

The typical way of connecting server and clients is by every client pushing requests to the server and the server pushing responses back to each client — the resulting SYS2D design is depicted in Fig. 4.3a. 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 **B2** and **C2** in Sec. 3.3). 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 **C3**). Both objections are reflected in the SYS2D constraints: the graph shown in Fig. 4.3a is sortable neither in horizontal nor in vertical direction.

Fig. 4.3b 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.

Publisher-Subscriber

A SYS2D interpretation of this layout can be seen in Fig. 4.3c: 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.

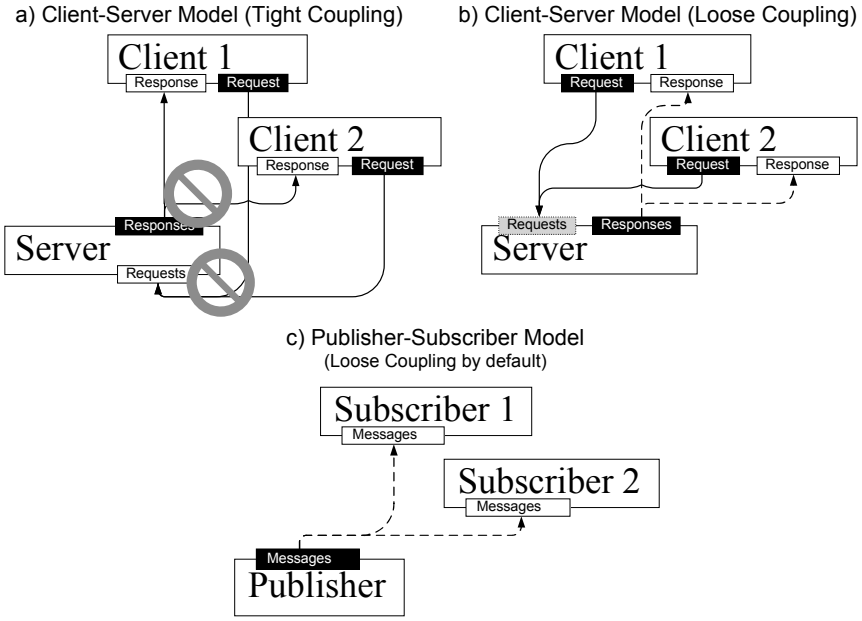

Figure 4.3: **Common interaction scenarios to illustrate and motivate the structural bias of** SYSTEMATICA 2D**.** – **a)** Tight coupled Server-Client layout, where clients push requests and the server pushes responses — this creates a build-order unsortability (a dependency loop) which makes incremental 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.

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 (e. g. [56, 57, 58]) — this is commonly referred to as lateral support. In a straightforward modeling of two units (see Fig. 4.4A), mutual modulation leads to a similar problem as discussed for the 'typical' Server-Client layout in Sec. 4.2.2, just that the sortability violation is in the 'processing flow' dimension.

We therefore propose an alternative layout, as shown in Fig. 4.4B: 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 properties: First, the graph is sortable again by avoiding the two-way modulation; by extension it also

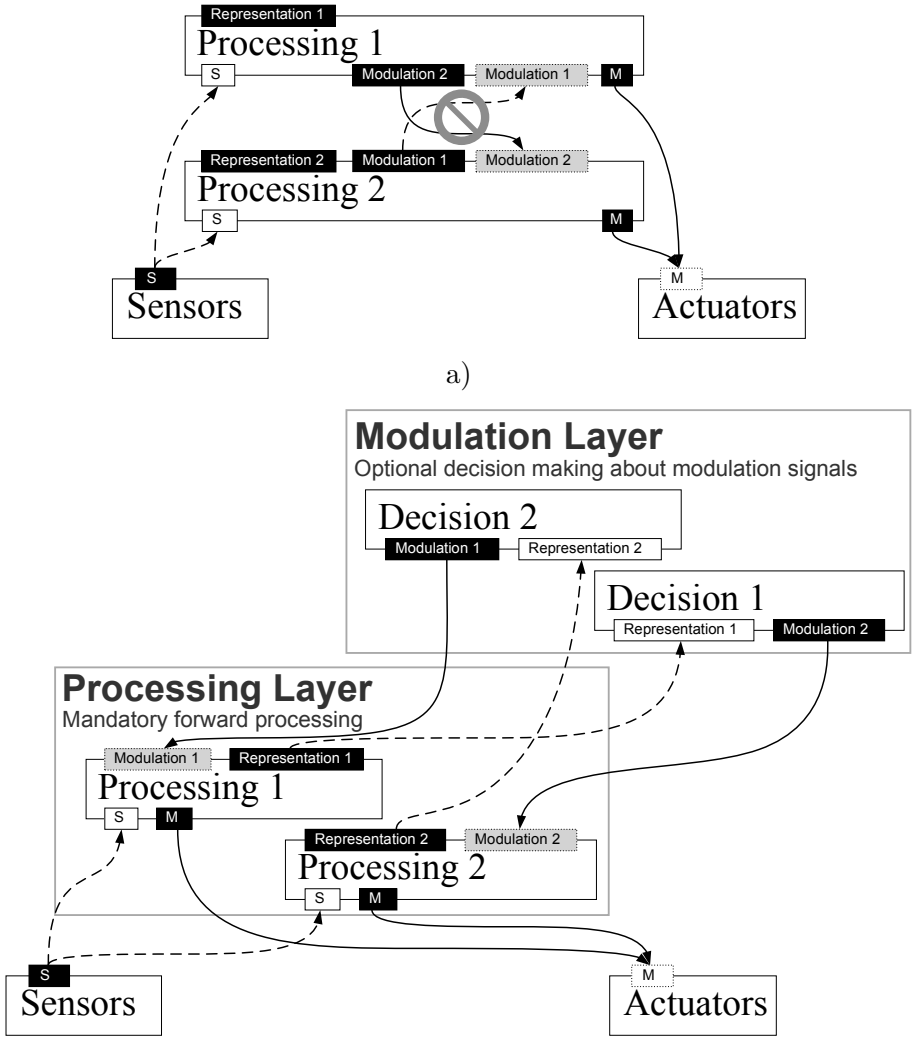

Figure 4.4: **Modeling of lateral support in SYSTEMATICA 2D – a)** Straightforward modeling of the lateral support pattern: the mutual modulation produces a processing flow unsortability. **b)** Model of lateral support from a SYSTEMATICA 2D point of view. To ensure separability, processing and decision, 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.

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

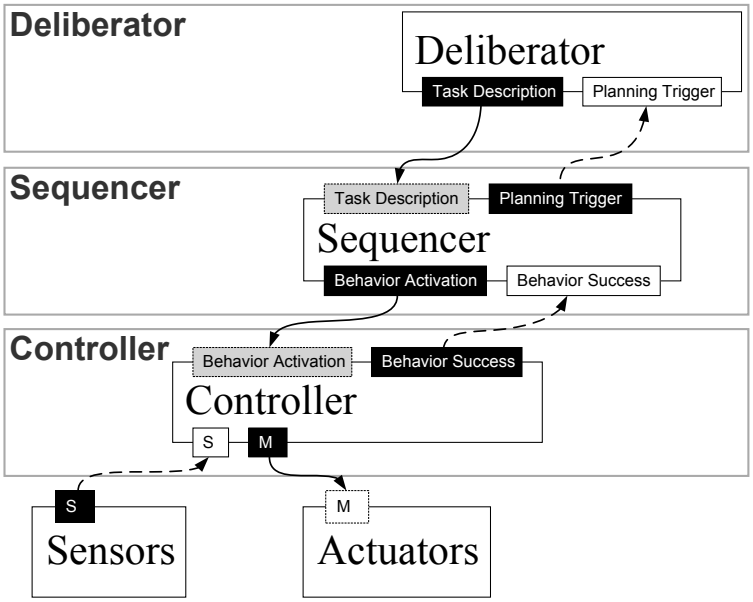

Figure 4.5: SYS2D visualization of the 3-Tier example from Fig. 3.3.

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).

4.3 Evaluation

After the definition of the SYSTEMATICA 2D system design and description language, we can now discuss the applicability of the language and the comparison to the existing formalisms evaluated in Sec. 3.4. To this end, we will present 'translations' of the evaluated formalisms 3-Tier, CogAff and SYSTEMATICA to the new language, a discussion of the full **AutoSys** SYS2D design will follow in Sec. 4.4.1 (a translation of the **ALIS** design to SYS2D is presented later in Ch. 7). Based on these examples and the functional constraints formulated in Sec. 4.2.1 we will then evaluate SYSTEMATICA 2D with the measure criteria presented in Sec. 3.3.

4.3.1 Translation of existing formalisms

Figures 4.5, 4.6 and 4.7 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

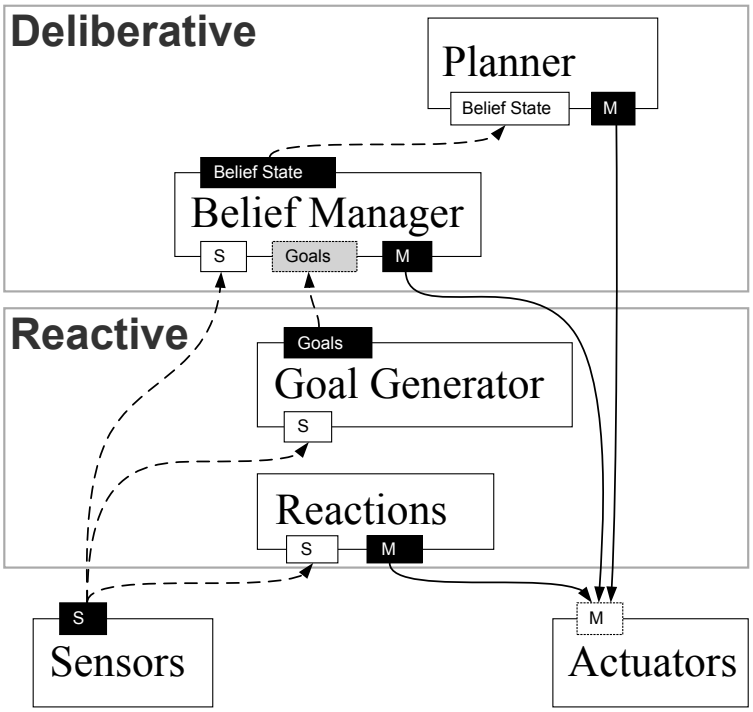

Figure 4.6: SYS2D visualization of the CogAff example from Fig. 3.4.

that these qualities are important in other notations as well. The visualization can therefore be done along the lines described in Sec. 4.1.4.

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. 4.5 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 [4]; 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.

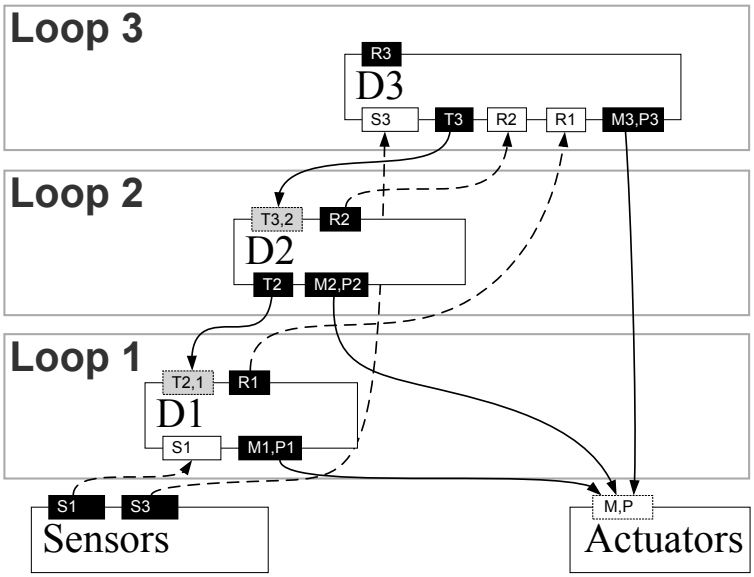

Figure 4.7: Sys2d visualization of a Systematica example similar to Fig. 3.5.

4.3.2 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 Sec. 4.2.2).

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 B2 (Decomposition to individuals). Beyond the functional units, a description of sub-architecture properties is also possible.

A3 (+) The formalism itself is mathematically formalized.

B1 (+) Interfaces are explicitly described by input and output ports.

B2 (+) The granularity can be matched to a per-scientist decomposition (see A2).

B3 (+) Coupling and dependencies are specified by input roles and push/pull connections.

C1 (+) The formalism can be translated to any infrastructure which supports communicating units and the three used input roles (see [54]) — which is possible in practically every infrastructure, from object-oriented platforms, over service- and blackboard-oriented systems to data-flow engines (see Ch. 5).

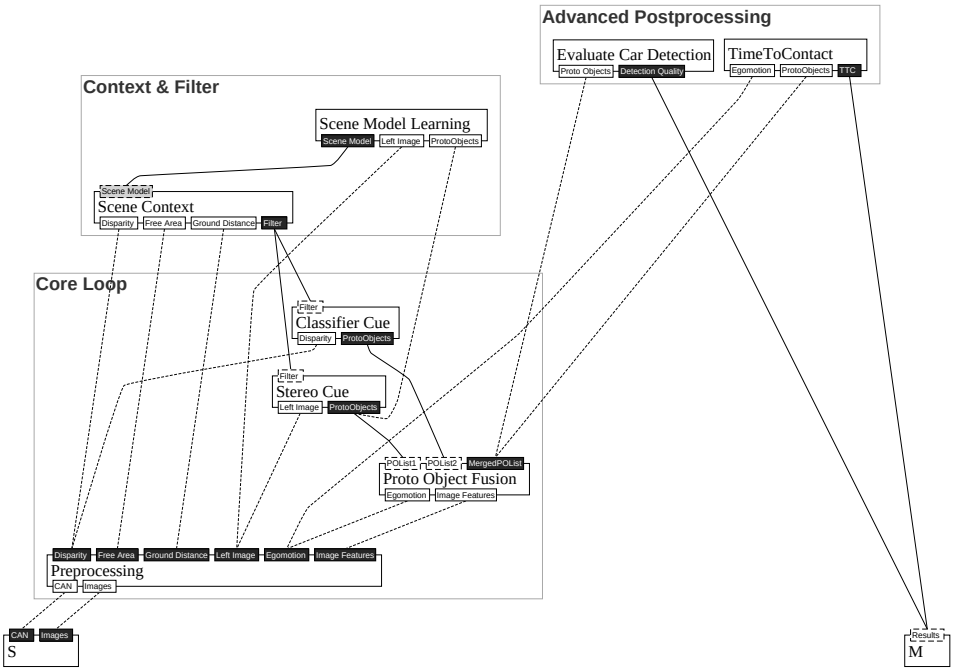

Figure 4.8: **SYS2D design of the AutoSys system** – The figure shows the complete design for **AutoSys**. All interfaces and dependencies are modeled, compare Fig. 2.3. Please see text for details.

C2 (+) The functional properties of incremental construction and complete subsystems defined in Sec. 4.2.1 aim directly at utilizing infrastructure properties of partial testing and graceful degradation.

C3 (?) The fine decomposition makes reusing single units possible without change (at least when observing B2). In addition, systems fulfilling the incremental construction constraint are very easy to decompose into reusable subsystems. However, currently only units or sets of units can be reused without a means of abstraction into larger building blocks (i. e.sub-architectures cannot be wrapped into complex units).

4.4 Results

4.4.1 The AutoSys design

AutoSys was the first system built completely according to a SYSTEMATICA 2D design. The final design can be seen in Fig. 4.8, for comparison to the reduced 'Boxes and Arrows'-like version chosen for publication see Fig. 2.3. It is apparent that

the main processing blocks introduced in Sec. 2.2 are each represented as one unit: Preprocessing (providing the sensory basis), Classifier and Stereo Cues (detecting regions of interest), Proto-Object Fusion (merging and stabilizing detections), Scene Context and Scene Model Learning (learning and applying filters to direct the cues' focus) and finally the two task-specific evaluations: detection of cars and estimation of time-to-contact.

In addition to the mandatory input (S) and output (M) units, the most important difference between the system sketch from Ch. 2 and the SYSTEMATICA 2D design is the specification of transmitted data as unit ports with specific input roles. From these input roles the ordering of units is derived. Since the reduced system sketch was created based on the SYSTEMATICA 2D design, it is not surprising that it reflects this ordering, with the exception that task-specific processing is arranged at the top to save space in the publication. A second significant difference between the two formalizations of the **AutoSys** system is the added description of sub-architectures in the SYS2D design, which in turn allows a specification of sensor and behavior spaces, as will be discussed in Ch. 6.

Motivation of Used Input Roles

We would like to discuss the choice of input roles at several positions in the design to illustrate their use. The use of 'Driving' inputs for sensory preprocessing results is straightforward, as is the use of the same type for the list of merged Proto-Objects used by task-specific post processing.

The Proto-Object Fusion unit uses 'DrivingOptional' inputs for the detections from individual cues. This indicates that the unit will operate with the cues present at the moment, but without depending on one specific cue to be present. The same is true for the use of filter input by the cues: if present, the filter will be used. All four of these 'DrivingOptional' inputs are not formulated as 'Modulatory' because data along these channels arrives synchronized to the sensory input (if it arrives at all) and thus is part of the processing flow: the Scene Context unit applies the current model to the sensory input to provide a filter *for each image* and the cues process the image and (potentially) the filter to provide detections *for each image*.

In contrast to this, the Scene Model Learning may have an internal dynamics or a long-term learning rule and is not assumed to provide a new scene model with every input image. Therefore, the provided scene model is received at a 'Modulatory' input by the Scene Context unit. As an example, the scene model contains a range of heights above the ground where detections are acceptable. Although this height

range may change slowly (Modulatory), the mapping from current ground distance and height range to acceptable positions in the image must be done for each new image (DrivingOptional).

Resulting System Details

The design itself shows that the **AutoSys** system *can be expressed and designed* in SYSTEMATICA 2D. The result is a *valid* SYS2D design, i. e. it satisfies all constraints in order to be sortable and allow incremental construction. This is not to say that the formalization process is a trivial one: interfaces and input roles have to be chosen to match each other and avoid violation of formal constraints — but as with most natural languages, the effort involved in following the grammar usually does not reduce the space of ideas that can be expressed.

The beneficial impact of using SYSTEMATICA 2D instead of currently existing system formalization languages lies in the information that can be *derived* from the design. We see three such derived details:

- **Unit Dependencies** resulting from the chosen input roles define which unit requires which other unit, such that the dependencies are minimal (only along Driving inputs) and have no loops (see proof for Sortability),
- **Build Order** resulting from the dependencies and their transitive partial order, meaning that there is a clear order in which components are to be integrated into the system,
- **Partial Test Graphs** resulting from the proof of complete subgraphs, meaning that during the integration of every unit, the minimal set of units necessary to test it is already present in the implemented system.

All three of these are derived from the theoretical, formal design of the system. They are results of the structural bias enforced by SYSTEMATICA 2D: specific input roles, sortability and incremental construction. However, all three are important not mainly for the design but for the construction of the system. As such, they demonstrate the ability of the SYSTEMATICA 2D language to enforce important constraints in the earlier design phase — without limiting the range of expression —, constraints which are then beneficial to the later construction phase.

A more detailed analysis of the possible decompositions of the **AutoSys** design, the impact this has on the implementation process and the reduction of time needed for system construction will follow in Ch. 7.

4.5 Conclusion

This chapter has introduced, discussed and evaluated the SYSTEMATICA 2D language. The main elements of the language itself are the *functional* system design, focusing on units, interfaces, input roles and connections, and the *descriptive* system design, focusing on sub-architectures and sensor / behavior space description. While this chapter has focused on the functional, a detailed analysis of the foundations and implications of these descriptive elements will follow in Ch. 6.

The input roles (Driving, DrivingOptional and Modulatory) as well as the separation of connections into push and pull are used to define dependencies between units along two independent dimensions: push/pull connections sort units according to *Build Order* (old-to-new), different input roles sort units according to *Processing Flow* (sensor-processing-actuator). Based on these dependencies, the formal constraints of *sortability* and *incremental construction* were defined, which led to proofs ensuring the completeness of a subgraph and the global deadlock-free operation of the designed system. These two constraints, combined in the term '*valid* SYS2D design', impose a significant structural bias on the design process, therefore their impact on the design of established software pattern was discussed.

To compare SYSTEMATICA 2D to the related notations introduced in Ch. 3, these designs were 'translated' into SYS2D designs and the new language was evaluated along the measure criteria. The ability to translate existing system approaches shows the expressiveness of SYSTEMATICA 2D. In addition, the fact that all translations resulted in valid SYS2D designs suggests that the constraints imposed are reasonable for the target domain of EI systems. Together with the expression of standard design patterns as valid SYS2D designs, this supports the claim that every EI system hypothesis can be rewritten in to a valid SYS2D design with the same functionality.

The evaluation of the new language shows that it satisfies most criteria well since it both allows flexible and standardized description in the design phase and enforces a structural bias which makes it very useful in the implementation phase. The two open points are the influence of the structural bias on domains other than EI and the missing ability to wrap sub-architectures into 'composite units' in order to abstract towards more complex systems rather just to reuse building blocks. Both of these will be the subject of future work.

In the last section of this chapter, the SYS2D design for **AutoSys** was shown and explained. The beneficial properties of using SYSTEMATICA 2D in the **AutoSys** system as compared to the ad-hoc design used for **ALIS** will be discussed in Ch. 7. In the next chapter, we will close the Hypothesis Text Cycle by presenting map-

pings from SYSTEMATICA 2D to common software infrastructure paradigms and thus answering the question how to get from a design to an implemented system.

5 From Design to System

A good design is by no means a guarantee for a working system. On the way to evaluating an intelligent system hypothesis (expressed by a design), the largest obstacle is the phase of implementation, usually done as a collaboration of many scientists and engineers. In short: a design is only as good as the system many people build from it – and a great number of obstacles in the implementation process may lead away from the design.

In this chapter, we will match the language elements of SYSTEMATICA 2D to the major software infrastructure paradigms (object-, service-, black-board-, data-flow-, and bus-oriented) to show that a careful design improves the implementation phase in two ways: First, it prevents or softens a set of recurring problems during implementation, such as high interdependency during testing as well as changes or delays in the finalization of units. Second, it allows a set of generic and (mostly) paradigm-independent system tools to be created such as reliable loose coupling of modulatory data or system-wide status monitoring.

This amount of preparation is not often attempted: knowing that the implementation phase is challenging, many projects focus their entire effort on integration and implementation rather than starting with a design. This is usually based on the assumption that such a design would consume precious time, change too frequently, and ergo not benefit the implementation phase at all. By looking at several infrastructure paradigms, reoccurring problems and possible solutions, we aim to show in this chapter that this is not the case.

We start by mapping SYSTEMATICA 2D, its elements, constraints and properties, to the most common infrastructure paradigms in Sec. 5.1. We then discuss a set of generic system elements which fit into this combination of design language and infrastructure and allow incremental construction and partial testing as well as system monitoring in Sec. 5.2. Finally, in Sec. 5.3 we present the resulting process for the full hypothesis test cycle, incorporating system formulation, implementation and revision.

5.1 Mapping Design To Infrastructure

Two questions will be answered in this section.

1. Given a SYSTEMATICA 2D design, what properties must a software infrastructure have to allow implementation of that system?
2. Given a SYSTEMATICA 2D design and a specific infrastructure paradigm, how does the design map to the infrastructure?

By starting with the first question we can evaluate different paradigms along these properties when we discuss their mapping.

5.1.1 Infrastructure Prerequisites

Both the set of criteria for formalization languages (Ch. 3.3) and the SYSTEMATICA 2D language itself (Ch. 4.1) have the expressed goal to improve the implementation process just as much as the design process. In SYSTEMATICA 2D, this improvement is based on a number of theoretical results of the structural bias, most notably incremental construction, partial testing and global deadlock-free operation. In order to bring any of these properties from the theoretical to the practical, it must be possible to map them into an infrastructure – and this mapping requires a set of infrastructure elements to be present.

As a first step, the SYS2D language elements have to be separated into descriptive and technically relevant constructs. Even though SYSTEMATICA 2D 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 Sec. 4.2.1. In Sec. 4.1.2 the differences between functional and technical language aspects were discussed. Thus, the necessary technical concepts are units, connections (without the push/pull distinction) and input roles (tightly or loosely coupled). 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.

The necessary infrastructure elements are derived mainly from the definition of these technical SYSTEMATICA 2D language elements (i. e. units and connections) and their interplay in terms of tight/loose coupling, incremental composition (for construction and testing) and subsystem decoupling for reuse:

Definition of Units and Interfaces

I1: The infrastructure must allow expressing functionality in terms of separate, self-contained processing units with a description of an interface in a standardized format – this will allow mapping SYSTEMATICA 2D units and ports.

Data Transmission between Units

I2: The infrastructure must provide a mechanism to transport (i. e. push or pull) data from one unit's interface to another – an essential prerequisite for any kind of system decomposition into units.

Independent Execution of Units

I3: The infrastructure must execute units independent from each other and from the (possibly non-trivial) processing of data transmissions – this allows both asynchronous processing, incremental composition (during construction) and decomposition (during reuse).

Support for the three SYS2D Input Roles

I4: The interfaces of units must allow 'Driving' (pull), 'Driving-Optional' (push/pull) and 'Modulatory' (push/pull) input roles with the associated hard/loose coupled behavior – this is the final constraint to validate the proven system properties in the implemented system.

Collaborative Development and Integration of Units

I5: The infrastructure must allow a number of scientists to collaborate, both in terms of unit development and in terms of unit interconnection, composition and decomposition – this is a requirement independent of any design language, simply for the purpose of allowing collaborative construction of more complex EI systems.

Some of these items will seem self-evident; we specify them anyway, both to arrive at a list which is as complete as possible and because mapping a SYSTEMATICA 2D design to an infrastructure paradigm can now be reduced to finding counterparts to each of these required elements I1-I5.

5.1.2 Mappings to Specific Infrastructure Paradigms

We will now apply the prerequisites to a set of infrastructure paradigms to show that they can serve as the basis for implementing any *valid* SYSTEMATICA 2D design, i. e. a design following the sortability and incremental construction constraints as defined in Sec. 4.2.1.

Object-oriented infrastructures

The class of object-oriented (OO) infrastructures is perhaps the most unconstrained, just as most other infrastructures discussed in the following are based upon an object-oriented language (such as Java, C# or C++) to establish their specific paradigm (e. g. all service-oriented infrastructures implement services, communication, routing, etc. as objects). As a result, the most we can say about requirements **I1-I5** is that all of them *can* be implemented in an object-oriented way: Objects serve as a straightforward mapping of units and the concept of interfaces is innate to most OO languages. Communication between objects can be done based on simple method calls or using a more elaborate communication library, e. g. CORBA or XML-RPC between processes. Independent execution of units can be done either using thread-based or process-/machine-based separation of objects, with the appropriate communication mechanism to support adding or removing units and their communication channels at run-time.

The implementation of the three input roles requires the possibility for objects to stop processing until they receive new data ('Driving' or connected 'DrivingOptional') and the possibility to accept modulation data asynchronously while waiting or processing ('Modulatory'). Since the latter mechanism is not trivial and also missing in most other infrastructures a generic solution will be presented separately in Sec. 5.2.1.

Last but not least, the ability for collaborative development is typically addressed by a separation of single unit development and the integration of units to (sub-) systems.

There is, to the best of our knowledge, no software infrastructure which is *purely* object-oriented, without exhibiting at least some traits of a more specific paradigm (the only candidate being Player[24], but even there the service-oriented aspects are not negligible). All of the previous remarks are equally true for the following infrastructure paradigms, simply because they are all based on some understanding of objects. By specifying some of the mentioned aspects, e. g. a specific understanding of objects and interfaces (e. g. a data flow) or a specific communication pattern (bus,

Systematica 2d	Player[24]	Microsoft Robotics Studio[14]
Units	Devices / Clients	Nodes (processes) / Services
Connections (push+pull)	Player Interface (local or networked)	DSSP Messages (local or networked)
Driving	implicit for Devices (upwards)	Port + Arbiter
DrivingOptional Modulatory	implicit for Clients and Devices (downwards)	Port + Mod. Switch

Table 5.1: **SYS2D language elements mapped to service-oriented infrastructures** – The table shows mapping of *technical* language elements of Systematica 2d to two service-oriented infrastructures. Player uses 'Devices' for composition of units (all of which run in the same process) and has an understanding of tight and loosely coupled communication very similar to Systematica 2d. Microsoft Robotics Studio allows arbitrary composition of services into nodes (processes) and has very flexible message-based communication channels that can be tuned to perform all three input roles.

blackboard), they define a specific way to build systems. We will now look at these specializations and how well the mapping of Systematica 2d designs can be done there for the case of service-, blackboard-, data flow- and bus-oriented infrastructures.

Service-oriented infrastructures

Composing an EI system out of interacting services has arisen as a popular concept in the robotics community because it makes adding and removing functionality to/from the system very easy (**I3**). The most popular infrastructures using this paradigm are Player[24] and the Microsoft Robotics Studio[14] (MRS). Both wrap each functionality into a service object with a clearly defined interface (**I1**) and provide the mechanisms for transmitting requests and responses between services (**I2**).

Integration of services can be done in two ways: In the case of Player, the designer specifies which service uses which other service to fulfill lower-level operations (e. g. an obstacle-avoidance service using odometry, laser and wheel control services). In the case of MRS, each service is responsible for finding the partner services where they wish to push data to or pull data from (supported by a service discovery service). Both approaches are compatible with the basic sortability constraint of Systematica 2d in that they require services to know only of services below them.

For the case of **I4**, the argument presented for object-oriented infrastructures also

SYSTEMATICA 2D	XCF[22]	CAST[23]
Units	Processes	Processes (sub-architectures) Components (units)
Connections (push+pull)	Active Memory (XML-RPC)	Working Memories (CORBA)
Driving	Subscription + sync.	Pull from Memory + sync.
DrivingOptional Modulatory	Query	Pull from Memory

Table 5.2: **SYS2D language elements mapped to blackboard infrastructures** – The table shows mapping of *technical* language elements of SYSTEMATICA 2D to two blackboard infrastructures. The separation into units is more coarse-grain in CAST since components with similar functionality are grouped into one 'sub-architecture' running in a separate process. Both infrastructures feature one or multiple shared data domains (the 'blackboards': for XCF called 'Active Memory', for CAST called 'Working Memory') which are used for communication between SYS2D units. In both cases, handling loosely coupled information is easier than synchronizing to Driving inputs.

holds: 'Driving' and 'DrivingOptional' inputs are straightforward, 'Modulatory' inputs can be realized using the mechanism laid out in Sec. 5.2.1 below. Collaborative development (**I5**) on the other hand is a direct result of the separation of systems into services and therefore needs no further customization.

Blackboard-oriented infrastructures

As a special case of communicating services, blackboard infrastructures provide one central data storage and access point across which all services exchange data (the 'blackboard'). Examples are XCF[22], and (to some extent) CAST[23].

As the blackboard is mainly a specific mechanism for communication, other aspects such as the separation of a system into units or services (**I1**) as well as the independent execution (**I3**) and collaborative development (**I5**) of services map in the same way as for service-oriented architectures.

By its very nature the blackboard provides a mechanism for data exchange (**I2**), thus the only remaining interesting question is the handling of input roles (**I4**). Typically, all communication through the blackboard is loosely coupled, in the sense that data is provided by one side but without enforcing its usage in any way. In order to implement a tightly coupled 'Driving' input role, the receiving (i. e. the pulling)

Systematica 2d	YARP[20]	ROS[18]	ToolBOS[25]
Units	Processes	Processes	Processes
Connections (push+pull)	YARP ports	ROS (XML-RPC)	BBDM / VFS
Driving	yarp-connect + sync.	Subscription + sync.	CML connections (implicit)
DrivingOptional Modulatory	Observer	Subscription	CML connections + Mod. Switch

Table 5.3: **SYS2D language elements mapped to data flow infrastructures** – The table shows mapping of *technical* language elements of Systematica 2d to three data flow infrastructures. The first two (YARP and ROS) use loose-coupled subscriptions by default and therefore require synchronization on Driving inputs. The last (ToolBOS) uses direct, synchronized communication by default and therefore requires the 'Modulation Switch' (see Sec. 5.2.1) to allow reliable loose-coupled inputs. All infrastructures support distribution of units to processes and communicate over some form of network protocol.

unit must internally add the logic to synchronize to the required data, i. e. to wait and continually poll the blackboard, or use a subscription mechanism, as provided by XCF. The implementation of 'DrivingOptional' needs the same kind of mechanism, coupled with the information if a potential sender is present at all and/or a timeout for falling back to loose-coupled operation. Finally, the easiest way to implement 'Modulatory' roles is to add an expiration date to the modulatory data when it is sent so that the receiver can asynchronously check for non-expired modulation data and incorporate it into the processing if it is present (see discussion in Sec. 5.2.1).

These relatively simple extensions turn the blackboard into a compatible channel of communication for Systematica 2d designs, albeit by ignoring some of the specific advantages of blackboard infrastructures (e. g. dynamic data lookup and inspection or easy structural plasticity, which we think contradict the purpose of design-first system implementation).

Dataflow-based infrastructure

The concept of designing systems as a set of units combined into a data flow can be found in YARP[20], ROS[18] as well as in the ToolBOS platform[25] which was used to implement both of the recurring example systems **ALIS** and **AutoSys**, presented in Ch. 2.

All data flow infrastructures implicitly bring an understanding of units, interfaces,

connections and data transmission (**I1**, **I2**). The ability for executing units in parallel is given to most, and with the rapidly increasing amount of multi-core or multi-processor computer hardware this percentage is sure to increase even more in the future. This satisfies part of **I3**, but the demand for *independent* execution also requires the infrastructure to be able to handle units dynamically joining or leaving the system (in order to allow incremental construction and graceful decay). Although not usually innate to data flow-oriented infrastructures, this property can usually be recovered by using separate operating system processes for units or sets of units and connecting them through proxy-units within each process responsible for data transmission.

The handling of the different input roles (**I4**) requires a mapping to synchronization mechanisms within the infrastructure. For the case of 'Driving' inputs, a mechanism executing one unit once new data from another unit arrives at such an input is the most basic, and thus the most commonly available one. Extending this to 'Driving-Optional' inputs is straightforward, given that the infrastructure can make a unit wait for new data given the input is connected ('Driving' case) and continue without waiting if the input is not connected ('Optional' case). The last type, 'Modulatory', is more demanding since it requires handling several circumstances, including permanent, initial or recurring absence of connected sending units and we again refer to the generic solution that will be given in Sec. 5.2.1.

Finally, the ability for collaborative development (**I5**) of EI systems is addressed differently by different infrastructures, but usually involves unit type libraries and/or hierarchical unit definitions – all of which is made easy by the fact that units can usually be developed separately by different scientists and then combined to form a given system.

Bus-oriented infrastructures

We will look at bus-oriented infrastructures as the last class because of their relevance for embedded intelligence in the automotive and small-scale robotics domains. In the former, the CAN bus is a long established standard and the AutoSAR initiative[59] is on the way to establish a common infrastructure for interacting (hardware-)components on a car. In the latter, several robot construction kits (e. g. 'Bioloids'[60] or 'Kondo KHR'[61]) use a bus architecture to connect sensors, actuators and one or more processing modules.

Bus-oriented systems are characterized by a set of hardware- or software-modules (establishing a separation into units – **I1**) connected by a single, shared communica-

tion bus (**I2**). Typically, modules publish their processing results onto the bus so that all other modules can read and process them, if they so choose. Bus-oriented systems are very well suited for dynamically adding or removing modules exactly because of this loose-coupled all-to-all communication and modules typically run completely independently because in most domains each module has a dedicated hardware (**I3**).

In terms of input roles (**I4**), bus-oriented systems behave similar to blackboard-oriented systems: data cannot be sent directly to another module but the receiving module has to actively listen for the sender to emit the data. Thus, tight coupled communication can be built along the same lines. Just for the case of modulatory information the lack of persistence of data within the bus makes the proposed approach infeasible, however the generic mechanism presented in Sec. 5.2.1 will be able to overcome this problem.

Finally, the collaborative development of different system parts (**I5**) is made very easy by the clear separation into separate modules. The only necessary central coordination is the assignment of distinct names or identifiers to all bus messages of all modules potentially connected to the system.

5.2 Generic System Elements

The mapping of Sys2d designs to software infrastructures has shown that most technical language elements can be translated quite easily to a variety of technical counterparts. One feature which Systematica 2d expects of the infrastructure is slightly harder to archive: reliable modulation. The ideal behavior of a 'Modulatory' input port is that it should immediately react to incoming modulation signals, use them as long as they are valid and fall back to default behavior if the sender of the signals fails or is removed. This requires an understanding of validity of modulation signals, which is usually considered a semantic quality of data by many infrastructures (an exception being XCF, a blackboard-oriented infrastructure, where the 'Active Memory' is able to keep track of the validity of stored signals). The following section will lay out the basic elements for solving this issue in a general way.

In addition, we want to show that reliable modulation is just one example of a generic system element for Systematica 2d systems. Such elements are implementable, simple additions to a system which are based solely on the structure of the Systematica 2d language, not on any specific design. The reliable modulation is such a generic element because it works with every Modulatory or DrivingOptional input port, independent of the function of this input in the specific system. As an

example for a second generic system element we will describe a 'Sys2d Monitoring System', which is able to analyze the 'heartbeat' of a running system based on Systematica 2d in order to identify failed units and, using the incremental construction constraint, detect the root cause of the problem, i. e. the smallest subset of failed units which can explain the failure of all other failed units.

5.2.1 Reliable Modulation

Modulatory inputs face several unique challenges which are not generally addressed in implementation infrastructures (see Sec. 5.1 above): Since modulatory data is usually pushed to these inputs from units built later, at the time of implementation of the receiving unit it may not be clear where this data is coming from or even how many units will send data to the port. Nonetheless, the unit should adapt its processing immediately once modulatory data arrives. Finally, it is often not favorable to apply a received modulation indefinitely, especially if the sending unit does not send new data or disappears from the system altogether (this is strongly related to the property of graceful decay). In short, *reliable* modulation means: the receiving port must handle no input data, continuous input data as well as suddenly appearing and disappearing input data from an arbitrary number of sources.

We propose to solve these requirements by the help of a 'modulation switch' as seen in Fig. 5.1, a component which receives all modulatory data of one port, analyzes their temporal behavior and sends a reliable modulation to the unit's processing. By placing the switch behind the port, and thus inside the unit, it is independent from the infrastructure paradigm chosen in that it does not depend on the paradigm-specific communication mechanism (only for the case of blackboard infrastructures the handling of this problem using the blackboard is preferable).

The first step to providing reliable data is to determine which of the connected modulation data is currently valid: this will prevent using empty data from units not yet added to the system and using outdated data from units already detached (deliberately or due to failure) from the system. Validity can be computed in many ways, of which we will discuss two: For sparse or irregular data it is necessary to embed an expiry date directly into the data, allowing an explicit check for expiration. For periodically sent data, validity can be computed by comparing the time since the last input signal from one specific sender, $\boldsymbol{\Delta t}$, to the typical temporal behavior $\boldsymbol{N^{\Delta t}(\mu, \sigma)}$ of data arriving from the sending unit. Here, $\boldsymbol{N^{\Delta t}(\mu, \sigma)}$ is a normal distribution with mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$ over acceptable $\boldsymbol{\Delta t}$ and can be specified or adapted at runtime based on the observed intervals between arriving

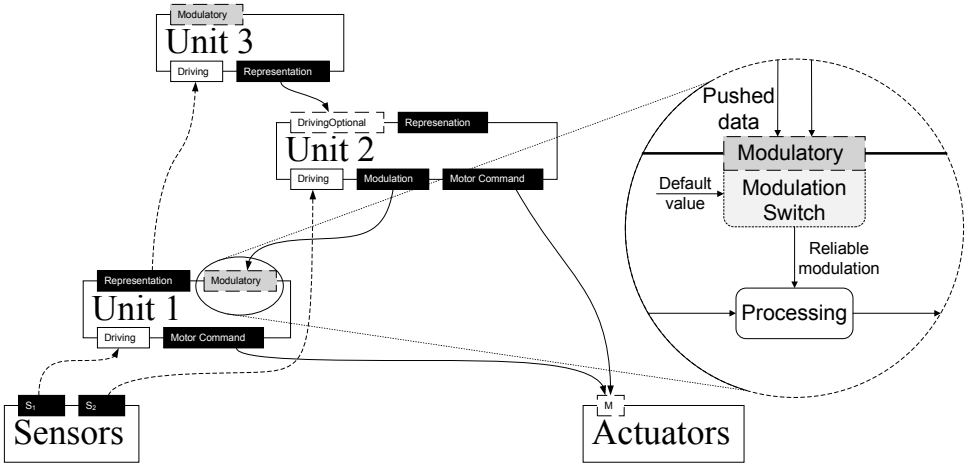

Figure 5.1: **Schema of the proposed modulation switch** – Situated within the unit and behind each modulatory input port, the switch determines the validity (see text for details) of received modulation data and selects or merges them into a reliable output, possibly falling back to a default value if no valid data is present.

modulation messages. Given a scaling factor $\boldsymbol{\nu}$ (in our systems, $\boldsymbol{\nu = 4}$), data can now be considered valid if $\boldsymbol{\Delta t \leq \mu + \nu\sigma}$.

Given the information about the validity of data from each sender, the switch can select from or merge all valid inputs: for some kinds of input an addition or multiplication of data is sensible, for others, especially for complex data types, a selection is more reasonable. One possible means for selection is the relative position of sending units along the processing flow (horizontal dimension), by selecting the modulatory data from the unit furthest to the right. Finally, in case none of the input data is valid, the switch must forward a default value to the processing.

This 'modulation switch' mechanism is similar to the 'suppression' mechanism introduced in the subsumption architecture[11], but it i) allows a clear definition of the modulatory interface (as part of the SYSTEMATICA 2D design), ii) accepts an arbitrary number of sending units and iii) adds a temporal adaptation mechanism.

5.2.2 SYS2D Monitoring System

One important aspect of system construction is debugging, i. e. the ability to detect and isolate failures in the system. This a problem with very many facets, from hardware failure to conceptual problems in the design. The one contribution to this issue that is most prominent on the system level is to detect when units fail and to decide whether they fail for internal reasons or because another unit failed.

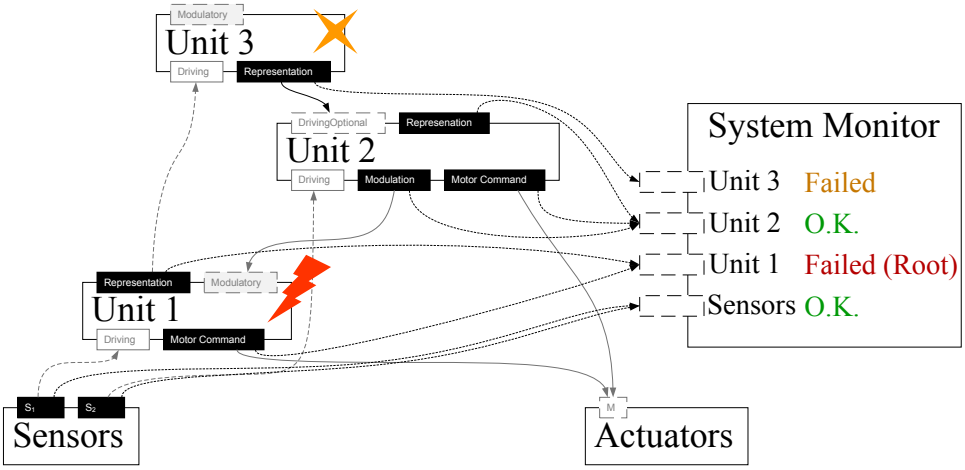

Figure 5.2: **Schema of the proposed SYS2D monitoring system** – The figure shows the proposed 'System Monitor' connected to all output ports of monitored units. By analyzing validity of received data (e. g. as done by the 'Modulation Switch', see Sec. 5.2.1), the monitor can identify failed units, i. e. units which are not running or send data at irregular intervals. In addition, the SYS2D design reveals the dependencies of units, allowing to identify whether a unit is responsible for the failure of other units ('root cause' analysis).

The mechanisms to check validity of incoming data packets presented for the Modulatory Switch can be adapted for this purpose: Assuming that any working unit produces output and that the validity of this output can be tested (either because it comes regularly or because it carries an expiration date), the decision whether a unit is working or not requires only the monitoring of all output of the unit. The proposed SYS2D Monitoring System (see Fig. 5.2) therefore requires access, i. e. pull connections, to all outputs of all monitored units (this implies that the motor control unit $\boldsymbol{u_2}$ cannot be monitored unless it produces some status output). Since the monitor does not depend on any specific data to arrive from the monitored units, it can be interpreted as a SYS2D unit with DrivingOptional inputs pulling data from all monitored outputs.

Using the proposed Modulation Switch at each of these inputs, the validity of outputs $\boldsymbol{o} \in \boldsymbol{O_i}$ of unit $\boldsymbol{u_i} \in \boldsymbol{U}$ can be determined using the mechanism described in Sec. 5.2.1: By adapting normal distributions $\boldsymbol{N_o^{\Delta t}(\mu_o, \sigma_o)}$ for all outputs, failure of a unit can be defined:

$$\boldsymbol{failed(u_i) \leftrightarrow \exists(o \in O_i) : \Delta t_o > \mu_o + \nu\sigma_o} \tag{5.1}$$

To decide whether a unit failed for internal reasons or not, the (*valid*) SYS2D design

has to be evaluated in addition to the state of each unit. The sorting relations and input roles of the design can answer the question whether all units $\boldsymbol{u_j}$ responsible for providing data to a Driving input of a failed unit $\boldsymbol{u_i}$ are functional. We can therefore define root failure of a unit:

$$\begin{aligned} \boldsymbol{root\_failed}(u_i) \quad \leftrightarrow \quad & \boldsymbol{failed}(u_i) \wedge \forall (u_j \in U) : \\ & (\exists (i, o') : (u_j, o', i) \in \mathbf{Pull}_i) \rightarrow \neg \boldsymbol{failed}(u_j) \end{aligned} \tag{5.2}$$

Thus, failed units can be split into two categories:

- **Root Failures** are units where all Driving inputs are supplied by functional units — in this case it is clear that the failure must be internal — and
- **Follow Failures** are units where at least one Driving input receives data from another failed unit — this does not imply that the unit is failure-free, but it is not surprising that it is currently not functional.

Thus, the combination of validity checks and formal design analysis in a System Monitor allows straightforward and generic detection of root unit failures.

5.3 Intermediate Summary: The Final Hypothesis Test Cycle

Up to this point, all necessary elements to support the hypothesis test cycle have been presented: the Systematica 2d language, including its motivation through the formalism measure, the functional properties derived from a structural bias on such designs, a mapping of all design elements to a large variety of infrastructures and finally a set of generic system elements to bring these systems to life. In this section we will revisit the steps of the original cycle, match the presented techniques to them and describe which properties improve them.

5.3.1 From Hypothesis to Design

The first step after the conception of a system idea or hypothesis is its formulation and successive refinement, leading to a design. We use the words ‘hypothesis’ and ‘design’ to refer to two distinct things: the former is the roughly shaped idea as it might appear in the head of a scientist or slowly takes shape in a discussion, the latter is the formalized, specified – to some level of detail – description of an EI system based on the hypothesis which allows to communicate to an outsider the essence of the hypothesis, shows important elements and, most notably, is the first step towards implementation and evaluation.

It is therefore not surprising that the process of getting from a hypothesis to a design in non-trivial. To give an example: for the **AutoSys** system, the initial hypothesis was "Multi-cue detection, fusion and scene context filtering can provide a superior automotive scene model." To get from this idea to the system design shown in Sec. 4.4.1 requires answering a phalanx of questions, including more detailed understanding what information cues detect, how and where the context filter operates, how it is learned and what information a promising fusion technique requires.

SYSTEMATICA 2D, as a formalization language, does not answer many of these questions. However, by providing a formal system description, it enforces answering all of them and in the process raises a set of new ones, most importantly the questions of dependencies between units. As a result, the final design is sure to actually contain all information necessary for system construction and evaluation – and this is not a feature inherent to most other design languages as shown in Sec. 3.4.

In addition, the designer does not have to consider the implementation process in general, as long as he or she considers the functional constraints for sortability and incremental construction: they encapsulate the structural bias necessary for easy implementation as seen in the following sections. These considerations can in turn be handled automatically by a design software (see Appendix A), thus giving immediate feedback during interactive design of the system.

5.3.2 Mapping Design to Infrastructure

Given a SYSTEMATICA 2D design representing the initial hypothesis, in most cases evaluations in the EI domain require an implementation of the design on a given infrastructure, which may or may not be connected to a physical artifact, like a robot or a car. This is usually a collaborative process and thus requires mapping of the design to future infrastructure elements so that they can be distributed and specific work areas can be defined. Important questions to answer here are what dependencies exist between these elements, in what time frame they can be / have to be constructed and how they can be tested.

A *valid* SYSTEMATICA 2D design, including sortability and incremental construction, provides answers to all of these questions. Dependencies can be directly derived from the sorting along the build order (vertical) axis, which in turn is derived from push/pull connections. If a unit is not related to another unit under the transitive hull $<_h^+$ then it is not depending on that unit, is not affected by changes to that unit and can be built before that unit. The second constraint of incremental construction together with the reliable modulation mechanism from the previous section

allows definition and independent execution of complete subgraphs, thus enabling the testing of units in a minimal but complete environment.

Finally, in case previously built systems are structured according to SYSTEMATICA 2D designs as well, reusing units (or complete subgraphs) is straightforward.

5.3.3 Stepwise Implementation of the Mapped Design

Based on the mapping and the dependencies of infrastructure elements to be implemented, the process of system integration is straightforward. Dependencies are clearly specified, reducing unintentional side-effects of local changes to units. Partial testing means that developers are not blocked by system elements which are not essential to testing their unit. Not even changes to the design during implementation are harmful, as long as they focus on extending the system and avoid drastically changing the sorting (horizontal and vertical) of existing units amongst themselves. The target system is constructed incrementally (given that it is a *valid* SYS2D system) and can be continuously tested. Naturally, the fact that systems are globally deadlock-free by definition helps to avoid a variety of time-consuming problems during this phase as well.

5.3.4 Evaluation and Publication of Implemented System

Once major parts of the system are operational it is possible to return to the initial problem, which is to perform experiments in order to evaluate the initial hypothesis.

One such experiment – and the one least influenced by the design – is the question whether the system as a whole performs as expected. Equally important for the argument of most hypotheses, however, is showing that it is the proposed new idea which is responsible for this performance. To return to the example of **AutoSys**: if the hypothesis is that multi-cue fusion and context filtering are crucial to scene modeling then a valid experiment is to run the system *without* context filtering or only with a single cue in order to show that these subsystems do not perform well.

The often mentioned property of partial testing, resulting from the structural bias in SYSTEMATICA 2D, allows such experiments by defining minimal but complete subsystems and allowing them to be run separately. The introduced modulation switch may even allow such adding or removing of system elements while the system is running, thus showing graceful degradation. Results of such experiments will be shown in Ch. 7.

5.3.5 Decomposition of Final System and Revision of the Hypothesis

After evaluating and publishing the implemented system hypothesis, the way to new ideas and new systems is open. On the one hand, decomposition of the implementation is useful for making well-working subsystems available to be used in other systems. Also, individual algorithms can be optimized or substitutes can be tested in the context of a full, integrated system, with the possibility for realistic data collection, full sensory-motor coupling etc.

On the other hand, the fact that the system is reflected in a formal design eases the development of variations of the original hypothesis, for instance, to stay with the **AutoSys** example, by accepting that a valuable preprocessing was achieved which can be integrated into future systems, and focusing on the next interesting challenges such as scene prediction or behavior learning.

5.4 Conclusion

This chapter has addressed the essential problem of bringing a theoretical Systematica 2D design to a practical, running system on a given software infrastructure. On the examples of popular software infrastructures for intelligent systems we presented how the technical language elements of Systematica 2D can be mapped to the concepts of each infrastructure and thus illustrated how a theoretical design can be turned into practice. The elements of a Sys2D design which are relevant for this mapping are units, connections and input roles. These were shown to match main concepts of many software infrastructure paradigms, with a focus on service-oriented (Player[24] and Microsoft Robotics Studio[14]), blackboard-oriented (XCF[22] and CAST[23]) and dataflow-based (YARP[20], ROS[18] and ToolBOS[25]) infrastructures. Thus, this mapping provides a straightforward way of translating the technical elements of a Sys2D design into a structure for system implementation.

One exception is the ability of a software infrastructure to provide means for reliable modulation, which we found to be present only in blackboard-oriented infrastructures. We therefore formulated a generic and practical method to handle loosely coupled inputs (i. e. DrivingOptional and Modulatory) so that these can be used also on the other software infrastructures. As a second example of such generic system elements, a system monitoring concept, based on the concepts of reliable modulation and the formal Sys2D property of incremental construction, was introduced.

Finally, in an intermediate summary of the previous chapters, both the mapping and the generic elements were put into the context of the completed hypothesis test

cycle to bridge the gap between theoretical hypothesis formulation and practical system construction. Given this improved process for the hypothesis test cycle resulting in formal designs on the one hand and in practical, measurable systems on the other hand, the following chapter will look closer at the properties and types of these systems and thereby attempt a first step towards understanding the space of possible systems.

6 System Properties and Types

Designs in SYSTEMATICA 2D are characterized by the set of units $\boldsymbol{U}$, including its two-dimensional ordering, and the additional description given by the set of sub-architectures $\boldsymbol{A}$. So far, in terms of system implementation and evaluation, only the technical aspects of $\boldsymbol{U}$, especially interfaces, dependencies and the ability for incremental construction (which is related to the order of units) were relevant. In this chapter we will discuss the meaning of sensor and behavior spaces, as specified in each sub-architecture $\boldsymbol{a_k} \in \boldsymbol{A}$, $\boldsymbol{a_k} = (\mathbf{name}, \boldsymbol{U_k}, \boldsymbol{S_k}, \boldsymbol{B_k})$, as well as the meaning of the global two-dimensional pattern of units and what can be deduced about the 'type' of design it expresses.

This discussion follows a set of global system properties and is based on the set of 'architecture properties' introduced by Goerick in [5]. These properties are discussed from scratch in the two-dimensional world of SYSTEMATICA 2D, where especially the layout of units allows more quantitative statements for the analysis of sensor and behavior spaces than the one-dimensional description of SYSTEMATICA.

Three system properties will be discussed in this chapter:

1. The *sensor and behavior spaces* must be defined in terms of their qualitative meaning for system description (why they should be described) and their practical interpretation in the system design (which design elements are responsible for implementing these spaces).
2. The question whether sensors are used to form *incremental representations* is important to distinguish the mentioned 'types' of design.
3. An analysis of sensor and behavior spaces and corresponding units will show the degree of *sensory and behavioral confinement* of each sub-architecture.

The introduction of these properties in Sec. 6.1 will provide a tool set which is then used to propose a non-exhaustive set of 'design types' in Sec. 6.2, including an overview of benefits and drawbacks and where existing systems fit into these.

6.1 System Properties

The properties discussed in this section create a relation between the technical, implementation-oriented features of the SYSTEMATICA 2D units $\boldsymbol{u_1..u_N}$ and the descriptive sub-architectures $\boldsymbol{a_1..a_M}$, especially the sensor spaces $\boldsymbol{S_1..S_M}$ and behavior spaces $\boldsymbol{B_1..B_M}$. The fact that units and sub-architectures are specified separately in any given SYSTEMATICA 2D design may be understood to imply that there is no direct connection between the two, but since all actual behavior of the full system stems from the implementation, ergo from the units, there ultimately is some form of redundancy in this description. The SYSTEMATICA 2D language does not make assumptions about the order in which design elements are specified, but we find it much more likely that the description of sub-architectures follows the description of units (or vice versa) than that both are independently and separately developed. Thus, it is legitimate to ask how unit interfaces, dependencies and two-dimensional order correlate with the sensor and behavior spaces, as well as with their interpretation in terms of incremental representations and confinement.

6.1.1 Sensor and Behavior Spaces

The reason for specifying the sensor and behavior spaces of a sub-architecture $\boldsymbol{a_k}$ is to give an impression which aspects of the physical (visual, auditory, tactile) or chemical world is consumed and which externally observable behaviors of the artifacts are produced by this specific sub-architecture. Following the definition from [5], a sensor space $\boldsymbol{S_k}$ is a subset of the full sensor space $\boldsymbol{S = S_e \times S_p}$, the combination of exteroception and proprioception, observed by the artifact. The behavior space $\boldsymbol{B_k}$ contains the externally observable trajectories through the state space of the artifact which are produced by motor commands or modulation information sent from $\boldsymbol{a_k}$.

Although $\boldsymbol{S_k}$ and $\boldsymbol{B_k}$ can be seen as theoretical, descriptive spaces, they are brought to life by the set of units $\boldsymbol{U_k}$ within $\boldsymbol{a_k}$. The question therefore is: which properties of these units create $\boldsymbol{S_k}$ and $\boldsymbol{B_k}$?

For the case of $\boldsymbol{S_k}$, the answer is relatively simple: SYSTEMATICA 2D defines a unit $\boldsymbol{u_1}$ to represent all sensors of the artifact by providing output ports with all sensory information available. In other words, the set $\boldsymbol{O_1}$, the set of output ports of $\boldsymbol{u_1}$, provides all information covered by the full sensor space $\boldsymbol{S}$. The sub-space $\boldsymbol{S_k}$ can now be derived from the set of sensor inputs $\boldsymbol{I_k \subset O_1}$ which are pulled directly to a unit $\boldsymbol{u_i \in U_k}$:

$$\boldsymbol{I_k} = \{o, o \in \boldsymbol{O_1} | \exists_{\boldsymbol{u_i \in U_k}} \exists_{\boldsymbol{i' \in I_i}} (\boldsymbol{u_1}, o, \boldsymbol{i'}) \in \mathbf{Pull_i}\} \tag{6.1}$$

Since $\boldsymbol{I_k}$ is very technical in nature and $\boldsymbol{S_k}$ has to be useful for descriptive purposes, a generalization from $\boldsymbol{I_k}$ to $\boldsymbol{S_k}$ is advised (e. g. $\boldsymbol{I_k}$={left image, disparity}, $\boldsymbol{S_k}$={visual information}), but there is still a very clear relation between the two.

For the case of $\boldsymbol{B_k}$, a similar argument based on the SYSTEMATICA 2D 'output unit' $\boldsymbol{u_2}$ and any data $\boldsymbol{O_k} \subset \boldsymbol{I_2}$ pushed there could be made, but the mapping is more complex for two reasons: First, units $\boldsymbol{u_i} \in \boldsymbol{U_k}$ can influence the behavior of the artifact not just by pushing motor commands to $\boldsymbol{u_2}$ but also by pushing data to 'Modulatory' or 'DrivingOptional' inputs of lower units $\boldsymbol{u_j} \notin \boldsymbol{U_k}, \boldsymbol{u_j} <_v \boldsymbol{u_i}$. Second, the behavior space $\boldsymbol{B_k}$ is intended to describe the *externally observable* behavior induced by $\boldsymbol{a_k}$, instead of the space of motor commands in $\boldsymbol{O_k}$. We see no formal method to bridge this gap and will therefore define a set $\boldsymbol{M_k}$, in addition to $\boldsymbol{O_k}$, with all information pushed from units in $\boldsymbol{U_k}$ to lower units above $\boldsymbol{u_2}$. The generalization from $\boldsymbol{O_k}$ and $\boldsymbol{M_k}$ to $\boldsymbol{B_k}$ now involves more theoretical consideration than from $\boldsymbol{I_k}$ to $\boldsymbol{S_k}$, but still has a clear grounding in the functional description of $\boldsymbol{U}$:

$$\boldsymbol{O_k} = \{\boldsymbol{i'}, \boldsymbol{i'} \in \boldsymbol{I_2} | \exists_{\boldsymbol{u_i} \in \boldsymbol{U_k}} \exists_{\boldsymbol{o} \in \boldsymbol{O_i}} (\boldsymbol{u_2}, \boldsymbol{i'}, \boldsymbol{o}) \in \mathbf{Push_i}\} \tag{6.2}$$

$$\boldsymbol{M_k} = \{\boldsymbol{o}, (\boldsymbol{u_j}, \boldsymbol{i'}, \boldsymbol{o}) \in \{\mathbf{Push_i}, \boldsymbol{u_i} \in \boldsymbol{U_k}\} | \boldsymbol{j} > \mathbf{2}\} \tag{6.3}$$

6.1.2 Incremental Representations

There are, in principle, two ways of constructing systems: either sub-architectures independently perform increasingly complex behaviors or they interact in order to use results of lower processing for higher functions. The first way can already be considered hierarchical. The quality of incremental representations, however, requires the ability of higher sub-architectures to create representations based on the existing ones from lower sub-architectures.[1]

The question whether a given system creates incremental representations cannot be answered by looking at the sensor and behavior spaces of the sub-architectures, but by analyzing the communication of units $\boldsymbol{u_i} \in \boldsymbol{U_k}$ with lower units of other sub-architectures. Since usage of lower representations is usually associated with 'Driving' inputs, this also leads to a horizontal ordering in most cases, i. e. higher units using lower representations are right of those lower units. The set of these used representations $\boldsymbol{R_k}$ can be formalized similarly to $\boldsymbol{M_k}$:

$$\boldsymbol{R_k} = \{\boldsymbol{i'}, (\boldsymbol{u_j}, \boldsymbol{o}, \boldsymbol{i'}) \in \{\mathbf{Pull_i}, \boldsymbol{u_i} \in \boldsymbol{U_k}\} | \boldsymbol{j} > \mathbf{2}\} \tag{6.4}$$

[1] This definition differs from [5], where incremental systems mean "systems where lower layers can already perform some meaningful behaviors without input from higher layers".

Thus, a system can formally be called to create incremental representation if $\exists_k \boldsymbol{R_k} \neq \emptyset$.

6.1.3 Sensory and Behavioral Confinement

The concept of sensory and behavioral confinement, as laid out in [5] can now be analyzed for SYSTEMATICA 2D systems by using the previously defined quantities $\boldsymbol{I_k}$ (the sensor inputs used by $\boldsymbol{a_k}$), $\boldsymbol{O_k}$ (the motor outputs produced by $\boldsymbol{a_k}$), $\boldsymbol{R_k}$ (the lower representations pulled by $\boldsymbol{a_k}$) and $\boldsymbol{M_k}$ (the modulatory signals pushed by $\boldsymbol{a_k}$).

Sensory confinement describes the total space $\hat{\boldsymbol{S_k}}$ of sensor information available to a sub-architecture $\boldsymbol{a_k}$, including the sensor inputs $\boldsymbol{I_k}$ directly observed and the sensor inputs observed through lower sub-architectures where representations of these are used in $\boldsymbol{R_k}$. For example, if $\boldsymbol{a_4}$ has no direct sensor input ($\boldsymbol{S_4} = \emptyset$) and only uses representations from $\boldsymbol{a_1}$ then the total observed sensor space is confined to $\boldsymbol{S_1}$. This analysis allows the designer to evaluate whether a given sub-architecture has access to the required sensor information (directly or through representations) and whether an extension of the directly observed sensor space would increase the total sensor input or not.

In the same way, behavioral confinement describes the total space of behaviors $\hat{\boldsymbol{B_k}}$ which a sub-architecture $\boldsymbol{a_k}$ can trigger, either by direct motor commands in $\boldsymbol{O_k}$ or by modulation $\boldsymbol{M_k}$ of lower units which in turn have control over the behavior space of their sub-architecture. For example, if only $\boldsymbol{a_1}$ has direct control of the motors ($\boldsymbol{B_1} \neq \emptyset, \forall_{k>1} \boldsymbol{B_k} = \emptyset$), then the total behavior space of all higher sub-architectures is confined to $\boldsymbol{B_1}$. This is usually not a desirable property since the behavioral demands of a growing and potentially learning system may not be known at the time of creation of $\boldsymbol{a_1}$, thus a confinement to $\boldsymbol{B_1}$ will limit the system's abilities. The obvious solution is to give higher sub-architectures some level of direct motor control[5].

6.2 EI System Design Types

The system properties discussed in the previous section are all grounded in the units and their connections, therefore the ordering of these units must reflect these properties to some extent. Although each specific system, based on a specific system hypothesis, is unique, we want to discuss three types of designs, each characterized by a combination of system properties and a closely related rough layout in the or-

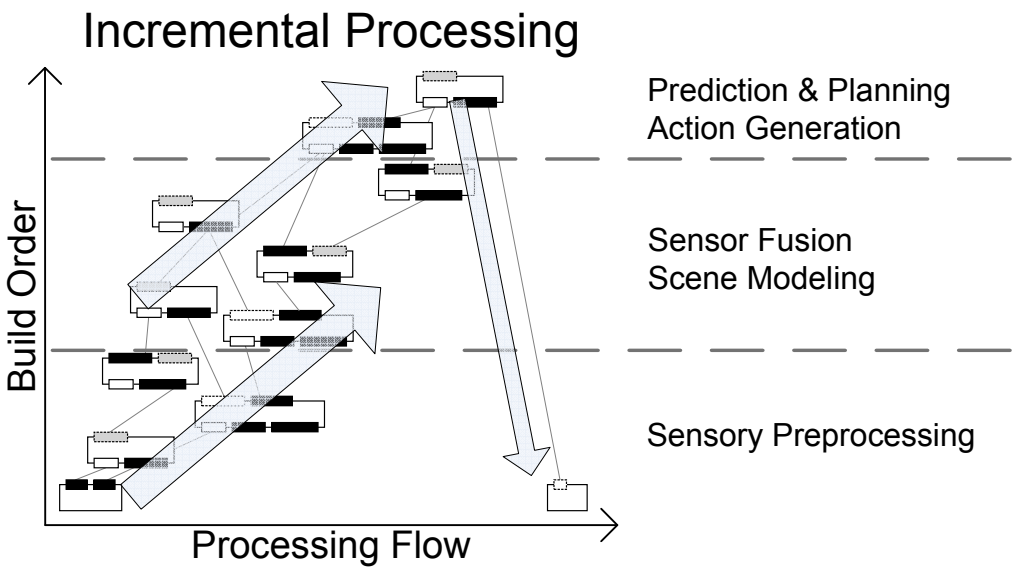

Figure 6.1: **Schematic of the first discussed system-wide integration approach** – (Pre-)Processing steps are incrementally added until the internal representation supports making complex decisions (reasoning, planning, ...), thus a motor command is produced by the top-most component.

dering of units. This set of types will not be exhaustive, but it does cover a large area as relations to existing systems will show.

6.2.1 Incremental Processing

The first discussed system type we call 'Incremental Processing' since it is found most commonly in systems with heavy sensory processing focus, usually following the Sense-Plan-Act principle or some variant.

As an example, a system of this type could start with sensory (e. g. visual) preprocessing, add cue fusion or another form of post-processing, proceed to scene analysis and scene modeling and finally add a form of reasoning, prediction and planning to derive motor commands. Thus, units are arranged along the secondary diagonal, from lower left to upper right and into the top left corner, see Fig. 6.1.

Integrations based on this approach will require a sophisticated sensory processing for the artifact to do anything at all: since the focus is on sensory processing, motor control is added at the latest stage. The danger in this approach therefore is that the integration process spends most available time on perfecting the pre-processing and this latests stage is only implemented in a rudimentary way, if at all. However, once

the sensory processing quality is achieved, it opens the door to powerful reasoning and planning.

Relation to System Properties

Two properties are mainly related to this kind of system. First, representations are very incremental: higher processing layers build on the representations of lower layers (such as image filtering, cue fusion results, etc.) – this chain of 'Driving' inputs leads to the arrangement of units on the secondary diagonal. Second, sensory confinement is very low to non-existent (access to sensors is essential for most units) but the behavior space of lower sub-architectures is small or empty ($\forall_{a_k,k<M} \boldsymbol{B_k} \approx \emptyset$).

Relation to Existing Systems

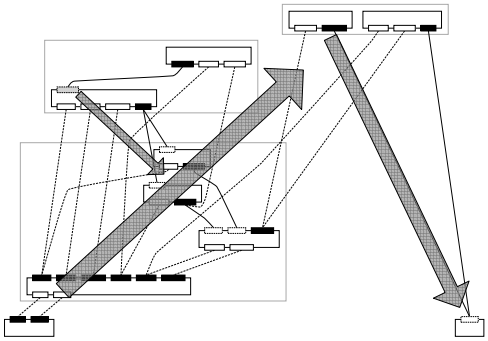

Figure 6.2: Schema of **AutoSys** design.

The 'Incremental Processing' system type can be found in many architectures mainly involved in sensory processing. The **AutoSys** design is a textbook example, as can be seen in Fig. 6.2. This is the case even though **AutoSys** contains a considerable amount of modulatory signals for context learning. However, the very late production of motor commands based on high-level sensory processing is the main indicator that this design type is used. Sec. 7.1.4 will discuss the categorization of **AutoSys** in more detail and compare it to the SYS2D design of **ALIS**, which can also be said to fall in this category.

6.2.2 Hierarchical Behavior

The second type of design discussed we call 'Hierarchical Behavior' and is the exact opposite of the one presented in the previous section. It is found in systems which have a strong focus on behavior generation, especially on the generation of increasingly complex behavior, as initially proposed in the Subsumption architecture[11].

As an example, a system of this type would start by adding simple behaviors like standing or walking, add more complex behaviors based on the simple ones, like searching and exploring, and proceed to even more complex, like search & rescue. As a defining characteristic, each behavior, or at least each set of behaviors of

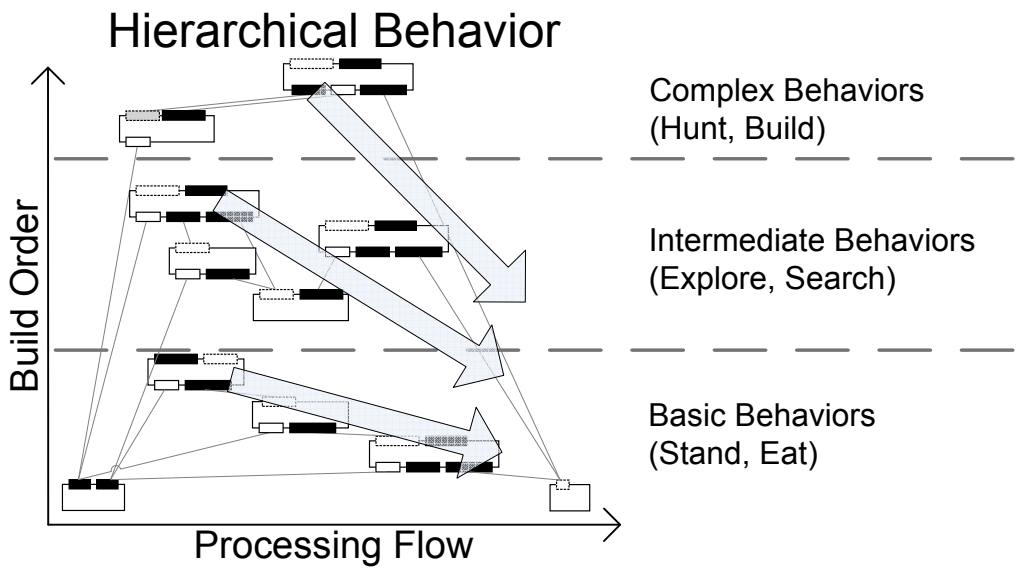

Figure 6.3: **Schematic of the second discussed system-wide integration approach** – Separate system layers perform increasingly complex behaviors. Each of these layers relies on a custom sensory pre-processing which is independent of other layers.

similar complexity extract the relevant information from the sensor independent of other behaviors. The result is a SYSTEMATICA 2D design which covers the whole two-dimensional plane: since units of different behavioral complexity are largely independent, they are distributed horizontally, see Fig 6.3.

Integrations based on this type of design require little coordination between different scientists working on different behaviors and still manage to achieve reliable and complex motor control. However, the quality of sensory pre-processing achieved by individuals or small groups, focused on one specific behavior, will typically fall far short of an incremental processing system as found in the previous type.

Relation to System Properties

In terms of system properties, the contrast to 'Incremental Processing' is obvious. First, sub-architectures do not use or produce incremental representations ($\boldsymbol{R_k} = \emptyset$), leading to a sensory confinement to the sensor directly provided to the sub-architecture. Second, behaviors are generated at every level, possibly even by modulating lower behavior, so the behavior spaces are not empty on any level.

Relation to Existing Systems

The 'Hierarchical Behavior' system type can be found mainly in architectures focused on behavior-oriented integrations, like the original Subsumption architecture[11] or the newer, behavior-centered EGO architecture[62].

6.2.3 Incremental Behavior

The last presented system type, which we will call 'Incremental Behavior', will try to find a compromise between the previous ones. Systems of this type create incremental representations, but at the same time use the rough, intermediate preprocessing results to create simple behaviors – until the higher representations are available and allow more complex motor control.

As an example, a system of this type could start with a rough (typically multimodal) sensory processing to enable a quick selection of basic behaviors (e. g. approach / retreat), potentially even including behavior learning. More sophisticated preprocessing and sensor fusion is added on top of that, necessitated by more complex tasks and behaviors, finally resulting in symbolic storage and reasoning as a necessary precondition for the most complex tasks. Thus, units form a pyramid of increasingly complex processing on the left and increasingly complex behaviors on the right, see Fig. 6.4.

Integrations based on this type will start with behavior generation (and possibly learning) as the first step. The danger in this approach is that the integration is forced to deal with toy scenarios for testing in the beginning and looses the focus on real world applications. Once a sufficient level of complexity is reached, however, all further processing layers are motivated and immediately evaluated in behaviors and thus have a behaviorally grounded frame of reference.

Relation to System Properties

The goal of this approach is to combine the favorable properties of 'Incremental Processing' and 'Hierarchical Behavior'. First, incremental representations provide each sub-architecture with the necessary combination of sensory input and/or information from lower units, removing sensory confinement. Second, the behavior generation on every level provides an increasingly complex, non-confined behavior space using direct motor control and/or modulation of lower units.

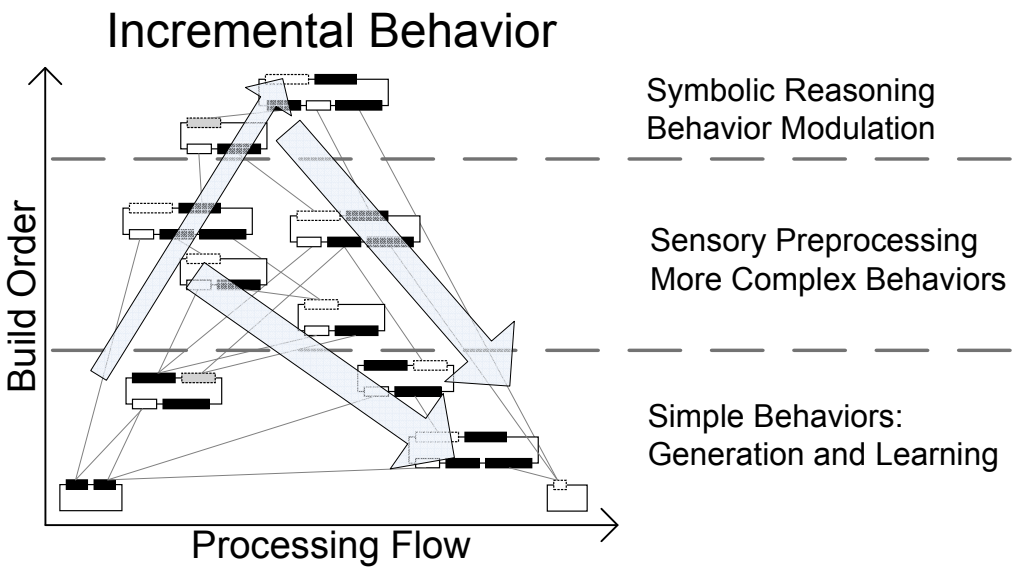

Figure 6.4: **Schematic of the third discussed system-wide integration approach** – Simple behavior generation is built first and then enriched / modulated by more complex preprocessing, decision making and behavior generation, motor commands are produced during all phases of development.

Relation to Existing Systems

There are, as yet, not many architectures which aim at this form of compromise between sensor-oriented and behavior-oriented designs. The only exception is the CogAff model[10] and its implementations[4], where the arrangement of system components in a two-dimensional grid spanning Sense-Process-Act on the horizontal and Reactive-Deliberative-Meta in the vertical seems to imply (pending the publication of running systems) that incremental representations in the 'Sense' and 'Process' areas can support increasingly complex behaviors in the 'Act' area.

6.3 Conclusion

In this chapter, the relation between functional and descriptive language elements of SYSTEMATICA 2D was formally introduced, leading to a definition of three system properties:

- 'Sensor / Behavior Spaces' characterize the information available to and influenced by a sub-architecture, based on the input-, output- and modulation-spaces of that sub-architecture,

- 'Incremental Representations', indicating the use of intermediate results (or 'representations') by higher units, based on the representation-spaces of sub-architectures and
- 'Sensory / Behavioral Confinement', determining to which level of sensors and actuators a sub-architecture can have access, based on the relation of all four previously mentioned spaces from one sub-architecture to another.

All these system properties can be evaluated for existing systems also without SYSTEMATICA 2D, as was done in [5]. However, when evaluating these rather theoretical properties for a SYS2D design, the 'type' of system (i. e. the specific combination of properties) has a specific impact on the sorting of units along the two SYS2D dimensions.

This relation between theoretical system type and visual system structure was demonstrated for three system types we find most interesting: 'Incremental Processing' (common for sensor-heavy systems like **AutoSys**), 'Hierarchical Behavior' (related to the Subsumption architecture[11], common in robotics) and 'Incremental Behavior' (a synthesis of both). For the first two, we discussed the defining system properties, the visual structure resulting from the unit ordering and examples among existing systems. The analysis of benefits and drawbacks led to the proposal of the third, 'Incremental Behavior', as a promising framework for future system hypotheses, since it seems to combine the benefits of incremental representation from 'Incremental Processing' with the behavioral grounding of 'Hierarchical Behavior'.

We are certain that these three system types do not cover the full space of possible system hypotheses, other combinations of system properties can easily be imagined. What we can say, however, is that the visual structure arising from the seemingly 'purely functional' description of units, together with the translation of existing notations to SYSTEMATICA 2D shown in Ch. 4, provide a powerful tool for system comparison and categorization.

7 Results

The central claims put forth in the introduction are that this work will present methods to improve:

1. **Hypothesis Formulation**, i. e. discussion, collaboration and publication of system hypotheses,
2. **System Construction**, i. e. the process of turning a design into a system, including handling collaboration as well as mapping to software infrastructure and
3. **Hypothesis Evolution**, i. e. reuse of theoretical concepts and system elements as well as comparison and extension.

The previous chapters have introduced the methods and analyses to solves these challenges, most prominent among them the Systematica 2d language. These methods can now be assigned to the claims in the following way:

1. **Hypothesis Formulation**: discussion, collaboration and publication are supported using the flexible, meaningful and standardized Sys2d description,
2. **System Construction**: implementation of a system is supported using improvements to collaboration during integration (especially incremental construction and partial testing) as well as by allowing an easy mapping of Sys2d designs to a variety of software infrastructures and
3. **Hypothesis Evolution**: reuse of the system is supported on the one hand using a formal analysis of system types to evolve and categorize the theoretical hypotheses behind each system and on the other hand by allowing decomposition and reuse of implemented system elements based on explicit dependencies in Systematica 2d.

This chapter will collect and extend the evaluations done in the previous chapters to sustain these claims. To address claims 1 and 3, Sec. 7.1 will demonstrate the expressiveness provided by Systematica 2d by applying the ability for translating existing formalisms (as shown in Sec. 4.3.1) to the **ALIS** system. Based on this translation, the system properties defined in Ch. 6 will be evaluated for both the

ALIS and the **AutoSys** systems, resulting in a categorization and a set of possible extensions for both systems.

To address claim 2, Sec. 7.2 will present experiments performed with the running **AutoSys** system to give an example for mapping a Sys2d design to a software infrastructure and to validate the system properties of incremental construction and graceful decay, both of which are vital for efficient system construction but also improve the system itself.

Finally, Sec. 7.3 combines all these qualitative measures to the single most important quantity for system construction: the effort it takes to go from system design to running system. By comparing the complexity and construction effort for a set of systems, some of which built with Systematica 2d and other without, we can show that Systematica 2d has a direct positive impact on this quantity.

7.1 Hypothesis Formulation and Evolution

Three aspects have to be investigated to validate the power for hypothesis expression of Systematica 2d: the ability to describe a variety of *abstract hypothesis concepts*, *existing systems* and *new systems*.

7.1.1 Description of Hypothesis Concepts

As for the first, the translations of existing EI approaches like 3-Tier, CogAff and Systematica presented in Sec. 4.3.1 are a good indicator that the range of system hypotheses which can be expressed is large enough to cover the currently most relevant ones. This is not a trivial observation because of the structural bias enforced by a *valid* Sys2d design, but all translations still led to such valid designs (see Fig. 7.1 for the original and translated form of a 3-Tier design). Finally, the reason for making the translations is not just to show the expressiveness of Systematica 2d but also allows a deduction of system properties and categorization into system types (see Ch. 6), thus allowing a founded comparison of system hypotheses.

Based on these investigations done in earlier chapters, the following sections will concentrate on the ability of Systematica 2d to describe existing systems (on the example of **ALIS**, which was not built based on a Sys2d design) and new systems (on the example of **AutoSys**). Finally, the system properties and types defined in Ch. 6 will be applied to both systems to compare them and derive possible extensions.

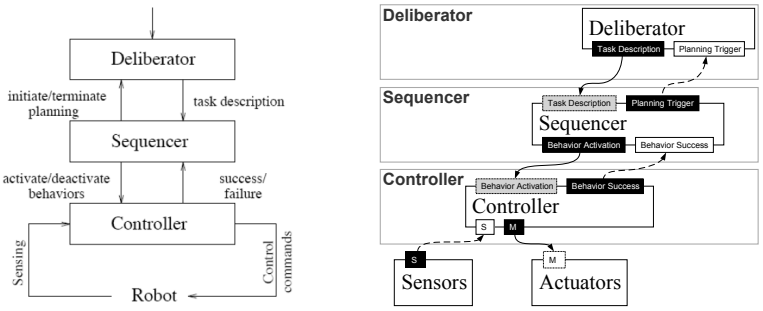

Figure 7.1: **Translation of hypothesis concepts** – The figure shows the original 3-Tier Description from [3] and the translated SYS2D Design.

7.1.2 Description of Existing Systems

The 'Autonomous Learning and Interacting System' (**ALIS**) was constructed in three phases (**ALIS**1[9], **ALIS**2[30, 28] and **ALIS**3[1]) over the course of three years. Although the central concept behind the three phases evolved, the systems themselves are quite different so only **ALIS**3 will be considered in this section. An introduction to the system itself, the underlying hypothesis and the major components is given in Ch. 2, together with a rough system notation, which is also shown in Fig. 7.2 in the upper left corner.

It is clear that this 'Boxes and Arrows'-diagram is not sufficient to produce a detailed, valid SYSTEMATICA 2D design, therefore additional material from [1] and [28] was used. The SYS2D version of the **ALIS**3 system can be seen in Fig. 7.2. It is not to be understood as a representation of the actual implemented system but as a realistic interpretation of the final system in the SYSTEMATICA 2D language. In the lower sub-architecture 'Reactive Loop', sensor information is preprocessed into Proto-Objects from which one 'active' Proto-Object is selected and used to create motor commands for interacting with an object presented by the tutor (either by gazing, pointing or walking). The higher sub-architecture 'Binding Loop' uses the features (e. g. position, size or motion) of the active Proto-Object together with recognized speech from the tutor to learn bindings between features, active behaviors and words or to create and validate expectations. Such expectations are triggered by recognized, previously learned words and are used to show confirmation (nod) or mismatch (shake head) for binding to Proto-Object features or to directly activate a bound behavior (e. g. grasping).

The main difference between the SYS2D design and the actual implementation is that DrivingOptional or Modulatory inputs could have been implemented (without

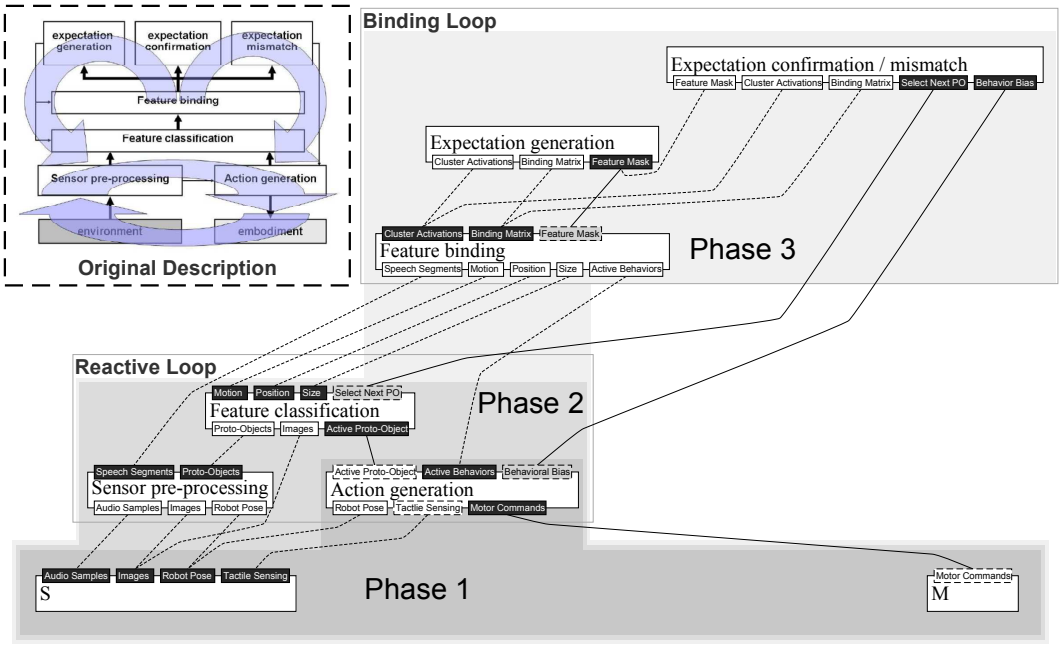

Figure 7.2: **SYS2D design of ALIS** – Top left corner: original **ALIS** notation, taken from [1]; Main figure: SYS2D design of the system, based on the original notation and additional description from [1] and [28]. The systems consists of a reactive and a binding loop, the former mainly responsible for preprocessing and action generation, the latter mainly responsible for learning semantic associations between features, behaviors and words. The shaded regions labeled Phase 1, 2 and 3 show possible complete subgraphs as they could be derived from the SYS2D design according to the definition in Sec. 4.2.1. Please see text for details.

affecting system behavior) as loosely coupled connections using the methods presented in Sec. 5.2.1, but since no explicit formulation of dependencies was done during system construction, most connections in the real system are tightly coupled. The benefit of such a dependency analysis is further demonstrated in Fig. 7.2 by the annotation of three complete subgraphs (labeled Phase 1, 2 and 3) that were derived from the design according to the definition in Sec. 4.2.1 and could have been used to incrementally build and test the system. The following section will revisit the **AutoSys** design from the same point of view and show incremental subgraphs. For the **ALIS** case this separation and the SYS2D design as a whole remain pure interpretations.

However, the purpose of this 'interpretation' of the existing **ALIS** system in SYSTEMATICA 2D is only partly to show that language flexibility is sufficient. Following the short review of the **AutoSys** SYS2D design, Sec. 7.1.4 will compare both designs along the SYS2D system properties defined in Sec. 6.1.

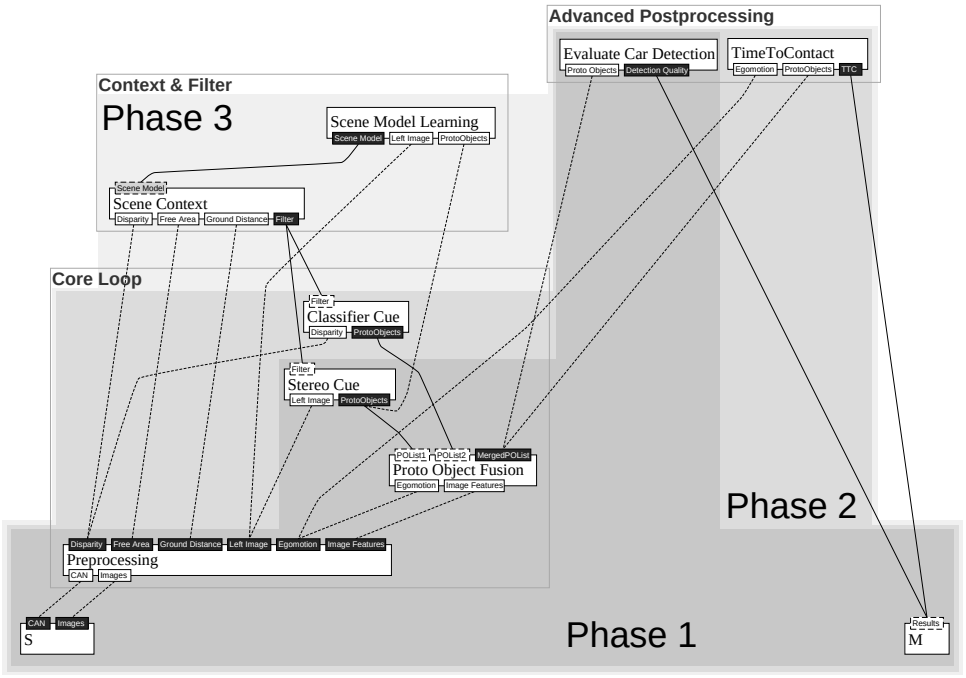

Figure 7.3: **SYS2D design of AutoSys with annotated subgraphs** – The figure shows the SYS2D design for the **AutoSys** system as already discussed in Sec. 2.2.2. Annotated are three complete subgraphs (labeled Phase 1, 2 and 3) which are derived from the design as defined in Sec. 4.2.1. These subgraphs were used during system construction to incrementally compose and test the full system. Please see text for details.

7.1.3 Description of New Systems

The full SYS2D design for **AutoSys** has already been introduced and discussed in Sec. 4.4.1. With the aim of validating the expressiveness of SYSTEMATICA 2D in this section, the focus will now be on the derived subsystems and their impact on system analysis and construction.

Figure 7.3 shows the design and three possible complete subgraphs following from the dependencies between units and the 'incremental construction' constraint. There are more such complete subgraphs of finer granularity: Technically, every unit in a *valid* SYS2D design has its own complete subgraph, e. g. the 'Classifier Cue' could be added to 'Phase 1' without the other units in 'Phase 2'. The decomposition into the specific three phases in Fig. 7.3 was chosen in order to allow experiments validating the properties of incremental construction and graceful degradation in Sec. 7.2.

In summary, we can say that the SYSTEMATICA 2D language was able to describe

relevant details of the **AutoSys** system in terms of system hypothesis, system decomposition and system construction. The subgraphs derived from the design were directly beneficial to both system construction and run-time properties in terms of incremental construction and graceful decay, as will be shown in Sec. 7.2. To discuss the final benefit of system categorization and comparison, the following section will apply the system types to the **ALIS** and **AutoSys** designs.

7.1.4 Comparison of ALIS and AutoSys System Properties

Based on the previous detailed presentations of SYS2D designs for **ALIS** and **AutoSys** we can now evaluate the SYSTEMATICA 2D language in terms of hypothesis comparison and evolution by applying the system types introduced in Ch. 6. Basic elements for this evaluation are the system properties of each sub-architecture: sensor/behavior spaces, incremental representations and sensory/behavioral confinement. Please note that there is no direct relation of these properties, defined for each sub-architecture, to the complete subgraphs, labeled 'Phase 1', 'Phase 2' and 'Phase 3', in both designs. Sub-architectures are defined by the designer while the complete subgraphs can be derived from the design at any time.

We will use the following nomenclature: all elements of the **ALIS** system will be denoted with a lower case $\boldsymbol{r}$ (for robotics), i. e. $\mathbb{S}^{\boldsymbol{r}} = (\boldsymbol{U^r}, \boldsymbol{A^r})$, and all elements of the **AutoSys** system will be denoted with a lower case $\boldsymbol{a}$ (for automotive), i. e. $\mathbb{S}^{\boldsymbol{a}} = (\boldsymbol{U^a}, \boldsymbol{A^a})$.

Starting with **ALIS**, the sub-architectures are:

$$\boldsymbol{a_1^r} = (\text{'Reactive Loop'}, \boldsymbol{U_1^r}, \boldsymbol{S_1^r}, \boldsymbol{B_1^r})$$
$$\boldsymbol{a_2^r} = (\text{'Binding Loop'}, \boldsymbol{U_2^r}, \boldsymbol{S_2^r}, \boldsymbol{B_2^r})$$

To specify **sensor and behavior** spaces, the sets $\boldsymbol{I_k^r}, \boldsymbol{O_k^r}, \boldsymbol{R_k^r}$ and $\boldsymbol{M_k^r}$ (sensor input, sensor output, representation and modulation) for $\boldsymbol{k = \{1, 2\}}$ can be derived from the graph:

$$\boldsymbol{I_1^r} = \{\text{Audio Samples}, \text{Images}, \text{Robot Pose}, \text{Tactile Sensing}\}$$
$$\boldsymbol{O_1^r} = \{\text{Motor Commands}\}$$
$$\boldsymbol{R_1^r} = \{\text{Speech Segments}, \text{Motion}, \text{Position}, \text{Size}, \text{Active Behaviors}\}$$
$$\boldsymbol{M_1^r} = \emptyset$$
$$\boldsymbol{I_2^r} = \boldsymbol{O_2^r} = \boldsymbol{R_2^r} = \emptyset$$
$$\boldsymbol{M_2^r} = \{\text{Select Next PO}, \text{Behavioral Bias}\}$$

From $\boldsymbol{I_1^r}$ and $\boldsymbol{I_2^r}$ it becomes apparent that the sensor space $\boldsymbol{S_2^r}$ of $\boldsymbol{a_2^r}$ is empty while the sensor space $\boldsymbol{S_1^r}$ covers all available sensors. The behavior spaces are harder to separate: although only $\boldsymbol{a_1^r}$ generates motor commands, $\boldsymbol{a_2^r}$ can influence the externally visible behavior using modulation. Based on the more detailed description of the system in [1] we can define:

$$\boldsymbol{S_1^r} = \{\text{visual}, \text{auditory}, \text{tactile}, \text{proprioceptive}\}$$
$$\boldsymbol{S_2^r} = \emptyset$$
$$\boldsymbol{B_1^r} = \{\text{gaze,point,nod,shake,approach,retreat,grasp}\}$$
$$\boldsymbol{B_2^r} = \{\text{nod,shake,approach,retreat,grasp}\}$$

Looking at **AutoSys**, we follow the same process:

$$\boldsymbol{a_1^a} = \{\text{'Core Loop'}, \boldsymbol{U_1^a}, \boldsymbol{S_1^a}, \boldsymbol{B_1^a}\}$$
$$\boldsymbol{a_2^a} = \{\text{'Context \& Filter'}, \boldsymbol{U_2^a}, \boldsymbol{S_2^a}, \boldsymbol{B_2^a}\}$$
$$\boldsymbol{a_3^a} = \{\text{'Advanced Postprocessing'}, \boldsymbol{U_3^a}, \boldsymbol{S_3^a}, \boldsymbol{B_3^a}\}$$

with:

$$\boldsymbol{I_1^a} = \{\text{CAN,Images}\}$$
$$\boldsymbol{R_1^a} = \{\text{Disparity,Free Area,Ground Distance, Left Image, Egomotion}\}$$
$$\boldsymbol{O_1^a} = \boldsymbol{M_1^a} = \emptyset$$
$$\boldsymbol{I_2^a} = \boldsymbol{O_2^a} = \boldsymbol{R_2^a} = \emptyset$$
$$\boldsymbol{M_2^a} = \{\text{Filter}\}$$
$$\boldsymbol{I_3^a} = \boldsymbol{O_3^a} = \boldsymbol{R_3^a} = \emptyset$$
$$\boldsymbol{M_3^a} = \{\text{Detection Quality,TTC}\}$$

Sensor spaces are therefore similar to **ALIS**: only $\boldsymbol{a_1^a}$ has access to the complete set of sensors. Behavior spaces are harder to define in the driver assistance domain since the result of the system is a visual signal to the driver, rather than a direct motor command. As the system is designed and implemented, this signal is produced solely by $\boldsymbol{a_3^a}$, thus we derive:

$$\boldsymbol{S_1^a} = \{\text{visual,proprioceptive}\}$$
$$\boldsymbol{S_2^a} = \boldsymbol{S_3^a} = \emptyset$$
$$\boldsymbol{B_1^a} = \boldsymbol{B_2^a} = \emptyset$$
$$\boldsymbol{B_3^a} = \{\text{car detections,time-to-contact}\}$$

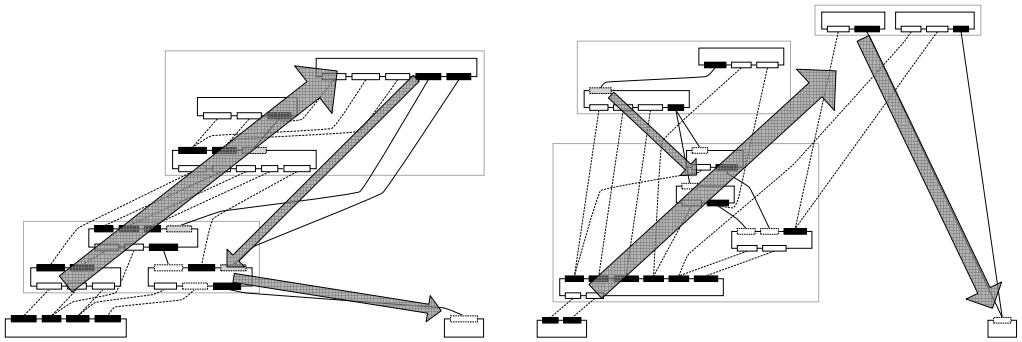

Figure 7.4: **Comparison of ALIS and AutoSys System Structures** – The figures show the **ALIS** (left) and **AutoSys** (right) SYS2D designs without text to allow an impression of the two-dimensional sorting of units based on the specified interfaces and dependencies. It is apparent that both systems perform incremental processing (thereby sorting units along the secondary diagonal from bottom left to top right), but in the case of **ALIS**, behavior generation is situated much lower than in the **AutoSys** design. This can be used for system type categorization, see text for details.

The second system property, **incremental representations** is easily answered looking at the $\boldsymbol{R^r}$ and $\boldsymbol{R^a}$ sets: both systems define higher sub-architectures using the representations generated by lower (in both cases: the lowest) sub-architectures.

Finally, **sensory and behavioral confinement** has to be evaluated: Sensory confinement is very low for both systems since the lowest sub-architecture produces a wide range of representations used by higher sub-architectures. Behavioral confinement exists for both systems, but in different ways. While in **ALIS**, $\boldsymbol{a_2^r}$ is confined to the behavior space of $\boldsymbol{a_1^r}$ because it has no direct motor output, both sub-architectures still have a non-empty behavior space. In contrast, the **AutoSys** system has a non-empty behavior space only for $\boldsymbol{a_3^a}$, all other sub-architectures are not able to trigger externally visible behavior (in this case: visualization) at all.

This difference in behavioral confinement is also apparent in the comparison view of both systems in Fig. 7.4. While the **ALIS** system closes the loop from sensor to actor early and adds the 'binding loop' purely for modulation, **AutoSys** incrementally adds processing modules and generates output from the highest sub-architecture.

In terms of **categorization**, Sec. 6.2 already classified **AutoSys** as a textbook example for 'Incremental Processing'. The **ALIS** system has characteristics of incremental processing, but with a behavior space for the lowest sub-architecture. Goal of the 'Incremental Behavior' system type was to combine incremental representa-

tions with behavior generation at every level, thus **ALIS** could be said to be between 'Incremental Processing' and 'Incremental Behavior'.

It is our strong belief that systems of the type 'Incremental Behavior' are more robust, flexible and, above all, able to ground all sensory processing in directly related behavior generation. To conclude the analysis of **ALIS** and **AutoSys**, we can therefore say that **ALIS** is closer to this goal than **AutoSys**. However, both systems could be extended to better allow behavior generation at all levels: In **ALIS**, the behaviors 'nod' and 'shake' are implemented in $\boldsymbol{a_1^r}$ although they are activated exclusively by modulation from $\boldsymbol{a_2^r}$ — moving these behaviors would naturally create the intended behavior hierarchy. In **AutoSys**, results are currently only generated based on the highest level of preprocessing — it might be possible to derive low-latency signals from lower sub-architectures, i. e. to alert the driver to close obstacles independent of their visual identity.

7.2 System Construction Process & Properties

In order to test whether the theoretical properties of incremental construction, partial testing and graceful decay apply to the implemented system as well, we will experiment with several decompositions of the implemented **AutoSys** in this section. These experiments will show the theoretical properties in action, with two implications for system construction: On the one hand, the ability to test subsystems eases collaboration, even if the results of a subsystem are not as good as those of the complete system (construction benefit). On the other hand, the same decomposition, coupled with the reliable modulation techniques introduced in Sec. 5.2.1, allows the complete system to continue operation if units in higher subsystems fail, even if this graceful decay slightly reduces system performance (operation benefit). Both properties are demonstrated by allowing units to be added and removed from the system at run-time.

As discussed in Sec. 4.4, the system was implemented on a dataflow-based infrastructure, observing the separation of units. All experiments are done with three specific subsystems of the whole system, which are depicted as 'Phase 1', 'Phase 2' and 'Phase 3' in Fig. 7.3. All of them use the detection of cars as a benchmark for an increasingly complex set of processing units, the time-to-contact computation is not used for these expriments. By running different subsystems independently from the rest of units, or even adding or removing units at run-time, we will show that the decomposition has the properties predicted by the SYSTEMATICA 2D design.

Figure 7.5: **Sample images with ground truth labeling used for AutoSys decomposition experiments** – The figure shows images from the stream used for evaluating the performance of **AutoSys**. White rectangles denote manually labeled cars used for measuring quality.

Experiments for this analysis were done on a data stream (recorded on a prototype vehicle) with additional hand-labeled ground-truth information. We chose a $70\boldsymbol{s}$ data stream, recorded at 10Hz during a typical drive on an inner-city road, with typical lighting conditions and occasionally missing lane markings, on a cloudy day, but without rain (see Fig. 7.5 for samples).

To evaluate system performance, we annotated the stream with ground-truth information, which can be seen in Fig. 7.5: for each image timestep $\boldsymbol{t}$, all task-relevant objects $\boldsymbol{gt_i(t) = (x_1, y_1, x_2, y_2), gt_i(t) \in GT(t)}$ were marked by a rectangular region of interest (ROI). The car detection unit produces a set of detection regions $\boldsymbol{v_j(t) = (x'_1, y'_1, x'_2, y'_2), v_j(t) \in V(t)}$.

By comparing the sets $\boldsymbol{GT(t)}$ and $\boldsymbol{V(t)}$, the set of ground-truth regions hit by a detection $\boldsymbol{GT^*(t) \subseteq GT(t)}$ and the set of detections which hit a ground-truth region $\boldsymbol{V^*(t) \subseteq V(t)}$ are computed.

We apply the standard measure 'quality' as proposed in [63] as $\boldsymbol{q(t) = TP/(TP + FP + FN)}$, with $\boldsymbol{TP = |GT^*(t)|}$, $\boldsymbol{TP + FN = |GT(t)|}$ and $\boldsymbol{FP = |V(t)| - |V^*(t)|}$, thus:

$$\boldsymbol{q(t) = \frac{|GT^*(t)|}{|GT(t)| + (|V(t)| - |V^*(t)|)}} \tag{7.1}$$

An ideal system would have quality values near 1. The **AutoSys** system has one cue (stereo) which is good at detecting many possible objects, of which not all are interesting (low $\boldsymbol{|V^*(t)|/|V(t)|}$, high $\boldsymbol{|GT^*(t)|/|GT(t)|}$) and one cue (classifier) which is very good at detecting interesting objects but misses the ones which do not look 'enough' like a car (high $\boldsymbol{|V^*(t)|/|V(t)|}$, medium $\boldsymbol{|GT^*(t)|/|GT(t)|}$). The Proto-Object fusion unit is responsible for finding a compromise between the two while the scene context filter helps to reduce the false positive output of both cues.

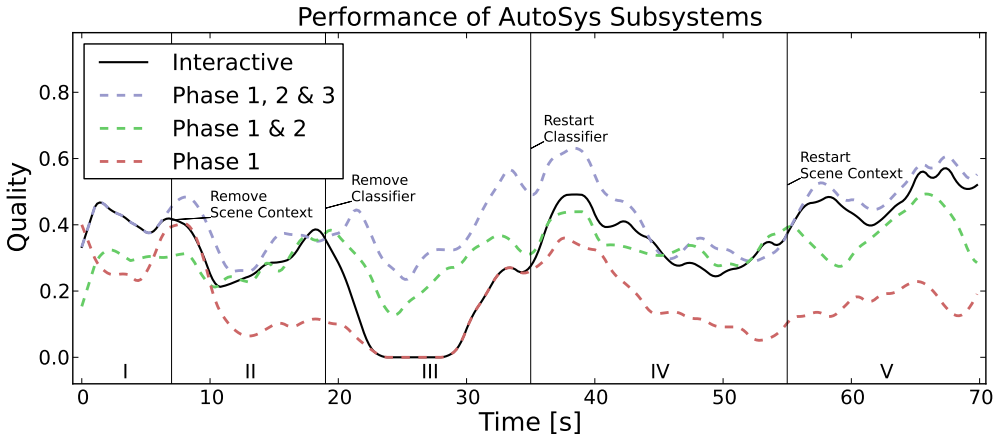

Figure 7.6: **Car detection quality for different AutoSys subsystems** – The figure shows the car detection performance recorded over four separate runs of the system. Three of them (dashed lines) were done with stable active subsystems. Since the quality shows considerable jitter in raw data, measurements have been smoothed. One interactive run (solid black line) demonstrates the system's ability for graceful decay and recovery. The system is started with 'Phase 3' running, after $7\boldsymbol{s}$ is falls back to 'Phase 2', after $19\boldsymbol{s}$ further down to 'Phase 1'. As can be seen, the performance drops to the level of the highest running subsystem. After $35\boldsymbol{s}$ the classifier cue is reactivated and after $55\boldsymbol{s}$ the scene context follows, leading to an almost complete recovery of detection quality.

We evaluate this quality once for the three functional subsystems, see Fig. 7.6. The results for the individual subsystems show the ability to test parts of the full system, as long as they adhere to the 'complete subgraph' property, as defined in Sec. 4.2.1.

The most interesting run of the system is the interactive run: in this case, the units of the system are manually shut down (simulating a crash) and restarted *during runtime*. The resulting performance shows that these crashes only reduce system performance to the performance expected of the smallest remaining subsystem. This is the result of loose-coupled input handling at the 'DrivingOptional' inputs for scene context filters and processing cues. Furthermore, the restart of units in the running system brings it back up to the performance of the full 'Phase 3'.

These experiments show that the theoretical properties of the SYSTEMATICA 2D design translate directly into design-time (partial testing) and run-time (graceful decay) properties of the implemented **AutoSys** system.

7.3 Reduced System Construction Effort

The previous sections have presented theoretical and experimental evidence for specific beneficial properties gained by using SYSTEMATICA 2D in the hypothesis formulation, implementation and evolution phases. One claim, which is independent of such specific properties and has not yet been supported by evidence is that using the new language and the related methods actually reduces the *effort* of the hypothesis test cycle as a whole.

This section will deal with this claim in the following way: The effort for hypothesis formulation and hypothesis evolution is extremely difficult to measure because it includes numerous unrelated activities like selecting a promising idea, project planning, writing publications etc. We will therefore focus on the effort spent on system implementation and compare it between systems built based on a SYS2D design and others built without. This comparison will require a definition of 'effort', in addition to an estimation of the 'complexity' of the compared systems since the effort clearly depends to a large part on the complexity of the built system.

Three robotics and three automotive systems are compared. On the robotics side, these are **ALIS**1[9] from 2007, **ALIS**2[30, 28] from 2008 and **ALIS**3[1] from 2009. On the automotive side, these are **FIRST**[64] from 2007, **SECOND**[54] from 2009 and **AutoSys** [2] from 2010. **ALIS**1, **ALIS**2, **ALIS**3 and **FIRST** were built without a SYS2D design, **SECOND** was built based on a design language which later evolved into SYSTEMATICA 2D and **AutoSys** was built completely based on a SYS2D design.

All six systems were built at the Honda Research Institute Europe under deliberately similar conditions: They were designed and developed by strongly overlapping groups of five to fifteen scientists and were all based on the ToolBOS software infrastructure[25]. On the one hand, this restricts generalization from this evaluation, on the other hand, these common points allow a comparison between systems and construction processes which would not be possible across institute or infrastructure boundaries.

The system complexity measure we use for comparison is directly based on this common basis for all the systems. ToolBOS is a dataflow-based infrastructure, so all systems are implemented as dataflow graphs composed of a certain number of processing module instances. The granularity of these instances as well as the number of new modules developed for each of the systems is comparable for all systems. The complexity of each system is therefore estimated by the number of processing module instances involved in each system's data flow graph.

To estimate the effort of system implementation, we recorded the number of weeks

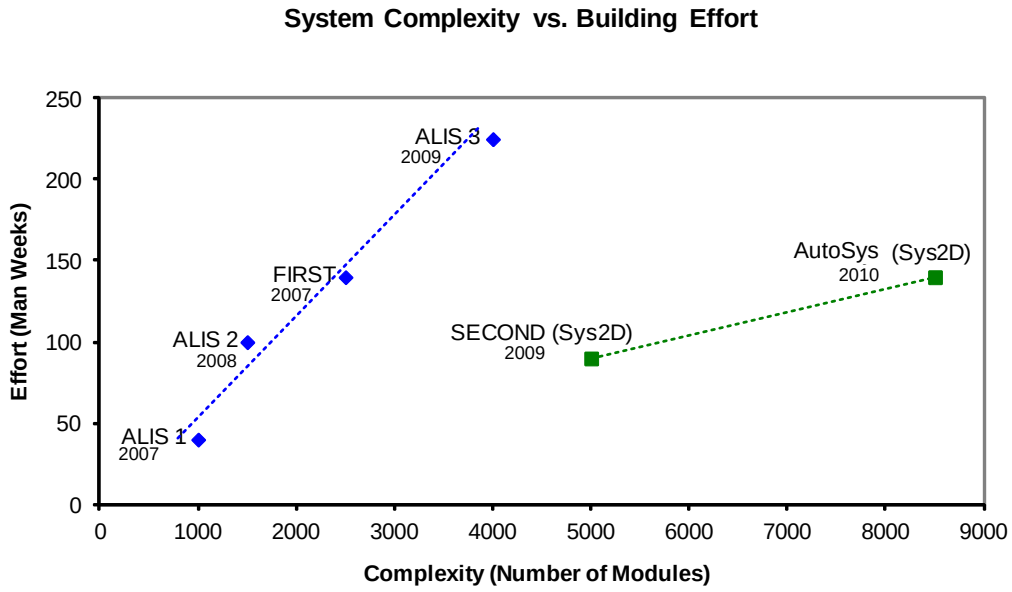

Figure 7.7: **Comparison of System Complexity and Implementation Effort for a set of existing systems** – The figure shows the system complexity and implementation effort for several existing EI systems. The systems in green were built based on Sys2d designs, in contrast to the systems in blue. As can be seen, the introduction of Systematica 2d significantly reduced implementation effort despite quickly growing system complexity. Please see text for details.

spent on each system construction process and multiplied it with the number of scientists involved, thus counting the number of 'man-weeks' per system. The result of the comparison of system complexity and implementation effort can be seen in Fig. 7.7.

First, it is obvious that system complexity grew dramatically — this growth is a result of increasing man-power available for construction combined with easy reuse of processing modules in the ToolBOS infrastructure. The growth of complexity is matched by a steep, proportional growth of implementation effort for the four Non-Sys2d systems (shown in blue). With the introduction of Systematica 2d (systems shown in green), the effort is reduced significantly despite continually increasing complexity, leading to a much lower effort-to-complexity ratio. Even without explicitly considering the specific system properties achieved by a Sys2d design, this reduction of implementation effort is a strong indicator for the utility of the language for intelligent system construction.

Two details of this study can be analyzed in more detail: One important question is whether the improved effort-to-complexity ratio can be explained simply by better

collaboration in the integration team, better software infrastructure or more reusable building blocks (besides SYSTEMATICA 2D). We believe that all of these can be ruled out because of the linear relation of complexity and effort over the first four considered systems (built without SYSTEMATICA 2D). If a gradual improvement of team or software infrastructure productivity had taken place, it would be visible in the relation of effort and complexity, and thus in a deviation from the linear relation, already for the first four systems. This makes the introduction of SYSTEMATICA 2D the only major difference in project organization.

The second question is which reasons the scientists involved in the system constructions give themselves for the improved construction speed. Since the study covers several systems which were built before the introduction of SYSTEMATICA 2D, a continuous user survey capturing main problems faced during each integration was not possible. However, discussions with scientists involved in both the **ALIS**3 and **AutoSys** integrations revealed two interesting observations: First, the possibility to test new algorithms in subsystems was considered the main improvement of the **AutoSys** integration over **ALIS**3, where testing new functionality usually required the whole system. This was made possible by the explicit description of dependencies in the SYS2D design and greatly reduced interference from simultaneous changes in the system. Second, several scientists indicated that the overall system decomposition, which was done initially based on the SYS2D design, did not affect their work during system construction, even though this decomposition basically remained stable over the whole system construction phase. We believe this shows that benefitial properties achieved because of the initial design do not require a detailed involvement of each participating scientist in the design process and are thus easily applicable to collaborative projects.

7.4 Conclusion

This chapter has collected and completed theoretical and experimental arguments to support the three basic claims of this work:

1. SYSTEMATICA 2D improves **hypothesis formulation** by providing a flexible and standardized language — the expressiveness is shown for the large-scale systems **ALIS** and **AutoSys** as well as on 'translations' of several existing cognitive models (CogAff, SYSTEMATICA and 3-Tier),
2. SYSTEMATICA 2D improves the **system construction** process by enforcing several beneficial properties using a structural bias during the design phase

— this results in a construction benefit (partial testing), an operation benefit (graceful decay) as well as an overall reduction in construction time,

3. SYSTEMATICA 2D improves **hypothesis evolution** i) by defining important system properties (incremental representations, sensory/behavioral confinement) which allow comparison and categorization of SYS2D and (through translation) Non-SYS2D systems and ii) by enabling easy system decomposition into reusable, functional subsystems (as was shown for **AutoSys**).

These properties, together with related methods introduced in this work, especially mapping to numerous software infrastructures, reliable modulation and easy system monitoring, allow large-scale, efficient and scientific hypothesis testing.

8 Conclusion

This work has presented a novel way to design, implement, compare and evolve system hypotheses for intelligent artifacts. Starting point and recurring motivation has been the improvement of the 'Hypothesis Test Cycle', i. e. the process from the first idea about a new system structure (the hypothesis) over the detailed design and the implementation in a physical artifact to the hypothesis comparison and evolution.

At the heart of the proposed set of methods is the new formal design language 'SYSTEMATICA 2D' which combines a specification of systems on two levels, the functional and the descriptive, with a carefully chosen set of constraints. The combination of these two achieves a variety of beneficial properties in all three phases of the hypothesis test cycle, from flexible and standardized design over collaboration-friendly and global deadlock-free implementation to hypothesis categorization and comparison.

The new language was compared to existing approaches from the domains of software engineering, software infrastructure for intelligent artifacts and intelligent system modeling by means of a set of measure criteria, proposed in this work explicitly for this purpose. The criteria allow evaluation of a language's ability to support the full hypothesis test cycle and were applied to four existing design languages. The evaluation showed that no current design language or software infrastructure is suitable for the full hypothesis test cycle.

The new language was then introduced in detail, including the formal set notation, a visual notation and the constraints of sortability and incremental construction which together form the elements of a *valid* SYS2D design. It was shown that the new formalism is able to express and enrich the evaluated modeling languages together with the two large-scale implemented systems **ALIS** and **AutoSys** and that the structural bias imposed by the constraints does not impair the flexibility of the language. Rather, several beneficial properties could be derived from these constraints, including the ability to decompose a design into complete subsystems to allow partial testing and incremental construction as well as proven global deadlock-free operation.

To provide evidence that the language is valuable for system implementation, a mapping of the relevant language elements to a wide variety of software infrastructures and -paradigms used for intelligent artifacts was detailed. In addition, the

analysis of the **AutoSys** system, built according to a SYS2D design, showed that implementation is not only possible but benefits from the system properties derived from the formal system description. Two generic system implementation techniques, reliable modulation and system monitoring, add to the utility of the language.

To complete the hypothesis test cycle, several properties were derived from SYSTEMATICA 2D to allow hypothesis categorization and comparison. By analyzing sensor and behavior spaces, incremental representations and sensory / behavioral confinement, different systems could be categorized into system types. As an example, three such types, 'Incremental Processing', 'Hierarchical Behavior' and 'Incremental Behavior' were defined based on combinations of system properties and related to existing systems, especially **ALIS** and **AutoSys**. This categorization is important for hypothesis comparison and, since possible improvements to bring a system from one system type to another can be derived, also for evolution.

All of these analyses together support the proposal that SYSTEMATICA 2D is able to significantly improve the hypothesis test cycle for intelligent artifacts and will allow faster and more precise development of future system hypotheses.

Outlook

Several elements of the language and the surrounding methods leave room for improvement: First, the definition of sub-architectures is not yet in a way to allow 'wrapping' them into complex units, which may be a prerequisite for future, much larger systems. Second, the mapping of system design to implementation would benefit from an automated code generation for selected infrastructures. Third, SYSTEMATICA 2D is an architecture description language in the software engineering sense and may, as such, allow application to a wider range of domains, given that it is formulated in a compatible way and the specific requirements of other domains are well analyzed.

However, we see the more interesting area for future work in the intelligent artifacts domain. Systems for intelligent artifacts, especially human-related robots and driver assistance systems, are being introduced at increasing speed, but without a common description or implementation language. By applying SYSTEMATICA 2D to existing and new hypotheses presented in these domains we hope to provide a better understanding of the directions and concepts followed at the moment, and, ultimately, propose improvements to take intelligent systems forwards.

Acknowledgements

First, I wish to thank my supervisors: Christian Goerick for his support, wisdom and fruitful discussions throughout the project, Sven Wachsmuth for his thorough review and constructive comments as well as Ulrich Rückert and Sebastian Wrede for their work on the examination board.

Second, the work on large-scale, real-world systems was possible only with the permanent support of the Honda Research Institute Europe GmbH and the many colleagues who participated in the builing of these systems. Their feedback, constructive input and open mindedness towards applying the design techniques put forwards here have made the work what it is. In particular, I would like to thank Bram Bolder, Antonello Ceravola, Mark Dunn, Nils Einecke, Jannik Fritsch, Alexander Gepperth, Mark-Oliver Gewaltig, Herbert Janssen, Jens Schmüderrich and Marcus Stein.

Last but not least, I am very grateful towards my fiancée Marta, my grandmother Helga, my parents, my sister and my friends for their love and support during the (sometimes strenuous) process of preparing this thesis.

Appendix

Visual Editing Software

Based on the visual representation described in Sec. 4.1.4, a software tool was developed to create, view and edit SYS2D designs (see Fig. A.1). Using the simple shapes of units, ports and sub-architectures, it is possible to quickly combine systems while automatically enforcing the sortability and incremental construction constraints and utilizing them 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 refer to the electronic publication of this thesis or send a short mail to `bdittes@googlemail.com`.

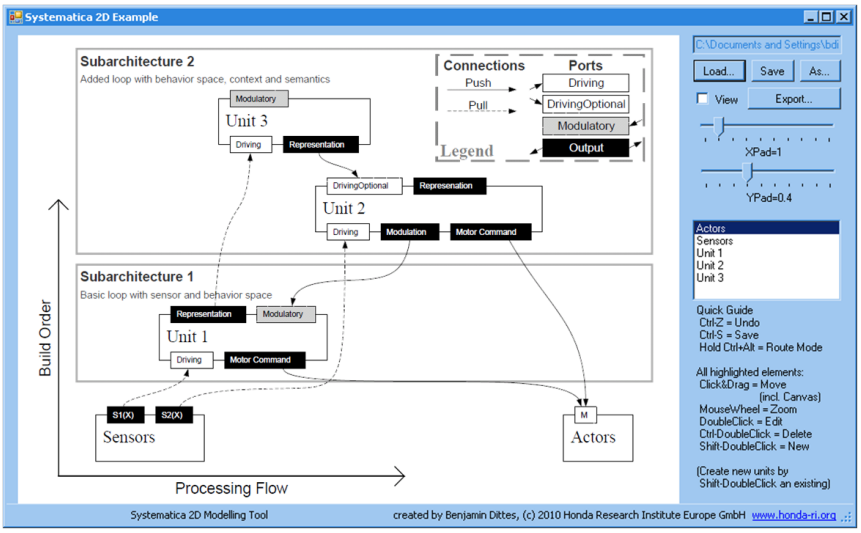

Figure A.1: **Screenshot of the SYS2D Visual Editing Software** – The tool allows editing of SYS2D designs, including all formal elements and validation of all constraints. See text for details.

Bibliography

[1] C. Goerick, J. Schmüdderich, B. Bolder, H. Janssen, M. Gienger, A. Bendig, M. Heckmann, T. Rodemann, M. Dunn, H. Brandl, X. Domont, F. Joublin, and I. Mikhailova, "Interactive online multimodal association for internal concept building in humanoids," in *IEEE-RAS International Conference on Humanoids 2009*. IEEE, December 2009.

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

[3] E. Gat *et al.*, "On three-layer architectures," *Artificial Intelligence and Mobile Robots*, 1997.

[4] 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.

[5] C. Goerick, "Towards an understanding of hierarchical architectures," *Autonomous Mental Development, IEEE Transactions on*, vol. 3, no. 1, pp. 54–63, March 2011.

[6] D. Roy, "A mechanistic model of three facets of meaning," *Symbols, Embodiment, and Meaning, de Vega, Glenberg, and Graesser, eds*, 2008.

[7] F. Hegel, C. Muhl, B. Wrede, M. Hielscher-Fastabend, and G. Sagerer, "Understanding social robots," *Advances in Computer-Human Interactions, 2009 (ACHI'09)*, pp. 169–174, 2009.

[8] D. Vernon, G. Metta, and G. Sandini, "A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computa-

tional agents," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 2, pp. 151–180, 2007.

[9] C. Goerick, B. Bolder, H. Janßen, M. Gienger, H. Sugiura, M. Dunn, I. Mikhailova, T. Rodemann, H. Wersing, and S. Kirstein, "Towards incremental hierarchical behavior generation for humanoids," *Intl. Conf. on Humanoids*, 2007.

[10] A. Sloman, "The cognition and affect project: Architectures, architecture-schemas, and the new science of mind," 2008.

[11] R. Brooks, "A robust layered control system for a mobile robot," *IEEE journal of robotics and automation*, vol. 2, no. 1, pp. 14–23, 1986.

[12] 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.

[13] Y. Shoham *et al.*, "Agent-oriented programming," *Artificial intelligence*, vol. 60, 1993.

[14] J. Jackson, "Microsoft robotics studio: A technical introduction," *Robotics & Automation Magazine, IEEE*, vol. 14, no. 4, pp. 82–87, 2008.

[15] H. Christensen, A. Sloman, G.-J. Kruijff, and J. Wyatt, Eds., *Cognitive Systems*. Springer Verlag, 2009. [Online]. Available: http://www.cognitivesystems.org/cosybook/

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

[17] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and 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] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *International Conference on Robotics and Automation*, 2009.

[19] S. Wrede, J. Fritsch, C. Bauckhage, and G. Sagerer, "An XML based framework for cognitive vision architectures," in *17th Intl. Conf. on Pattern Recognition*, 2004.

[20] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: Yet another robot platform," *International Journal on Advanced Robotics Systems*, vol. 3, no. 1, pp. 43–48, 2006.

[21] A. Gepperth, J. Fritsch, and C. Goerick, "Cross-module learning as a first step towards a cognitive system concept," *1st Intl Conf on Cognitive Systems*, 2008.

[22] J. Fritsch and S. Wrede, "An integration framework for developing interactive robots," *Software Engineering for Experimental Robotics*, pp. 291–305, 2007.

[23] N. Hawes, M. Zillich, and J. Wyatt, "BALT & CAST: Middleware for cognitive robotics," *School of Computer Science Research Reports – University of Birmingham CSR*, vol. 1, 2007.

[24] B. Gerkey, R. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th international conference on advanced robotics*, 2003, pp. 317–323.

[25] A. Ceravola, M. Stein, and C. Goerick, "Researching and developing a real-time infrastructure for intelligent systems - evolution of an integrated approach," *Robotics and Autonomous Systems*, vol. 56, no. 1, pp. 14–28, 2007.

[26] J. Schmüdderich, H. Brandl, B. Bolder, M. Heracles, H. Janssen, I. Mikhailova, and C. Goerick, "Organizing multimodal perception for autonomous learning and interactive systems," in *8th IEEE-RAS International Conference on Humanoid Robots, 2008. Humanoids 2008*, 2008, pp. 312–319.

[27] M. Heckmann, H. Brandl, J. Schmüdderich, X. Domont, B. Bolder, I. Mikhailova, H. Janssen, M. Gienger, A. Bendig, T. Rodemann, M. Dunn, F. Joublin, and C. Goerick, "Teaching a humanoid robot: Headset-free speech interaction for audio-visual association learning," in *The 18th IEEE International Symposium on Robot and Human Interactive Communication*, 2009, pp. 422–427.

[28] I. Mikhailova, M. Heracles, B. Bolder, H. Janssen, H. Brandl, J. Schmüdderich, and C. Goerick, "Coupling of mental concepts to a reactive layer: incremental approach in system design," in *Proceedings of the 8th International Workshop*

on Epigenetic Robotics, Brighton, England. Lund University Cognitive Science Studies 117, 2008.

[29] B. Bolder, M. Dunn, M. Gienger, H. Janssen, H. Sugiura, and C. Goerick, "Visually guided whole body interaction," *Robotics and Automation, 2007 IEEE International Conference on*, pp. 3054–3061, 2007.

[30] B. Bolder, H. Brandl, M. Heracles, H. Janssen, I. Mikhailova, J. Schmüdderich, and C. Goerick, "Expectation-driven autonomous learning and interaction system," in *IEEE-RAS International Conference on Humanoid Robots*, 2008.

[31] A. Clark, "Feature-placing and proto-objects," *Philosophical Psychology*, vol. 17, no. 4, pp. 443–469, 2004.

[32] R. Brooks, C. Breazeal, M. Marjanović, B. Scassellati, and M. Williamson, "The cog project: Building a humanoid robot," *Computation for metaphors, analogy, and agents*, pp. 52–87, 1999.

[33] F. Orabona, G. Metta, and G. Sandini, "Object-based visual attention: a model for a behaving robot," in *CVPR*, 2005.

[34] T. Michalke, R. Kastner, M. Herbert, J. Fritsch, and C. Goerick, "Adaptive multi-cue fusion for robust detection of unmarked inner-city streets," *IEEE Intelligent Vehicles Symposium*, 2009.

[35] N. Einecke, S. Rebhan, V. Willert, and J. Eggert, "Direct surface fitting," in *International Conference on Computer Vision Theory and Applications*, 2010.

[36] H. Wersing and E. Körner, "Learning optimized features for hierarchical models of invariant object recognition," *Neural computation*, vol. 15, no. 7, pp. 1559–1588, 2003.

[37] C. Igel, T. Suttorp, and N. Hansen, "Steady-state selection and efficient covariance matrix update in the multi-objective CMA-ES," in *Evolutionary Multi-Criterion Optimization.* Springer, 2007, pp. 171–185.

[38] C. Goerick, D. Noll, and M. Werner, "Artificial neural networks in real-time car detection and tracking applications," *Pattern Recognition Letters*, vol. 17, no. 4, pp. 335–343, 1996.

[39] N. Hawes and J. Wyatt, "Engineering intelligent information-processing systems with CAST," *Adv. Eng. Inform.*, vol. 24, no. 1, pp. 27–39, 2010.

[40] B. Dittes and C. Goerick, "A language for formal design of embedded intelligence research systems," *Robotics and Autonomous Systems*, vol. 59, no. 3-4, pp. 181 – 193, 2011.

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

[42] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and analysis of system architecture using Rapide," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 336–354, 1995.

[43] D. Garlan, R. Monroe, and D. Wile, "Acme: An architectural interconnection language," Technical Report, CMU-CS-95-219, Carnegie Mellon University, Tech. Rep., 1995.

[44] M. Scheutz and 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.

[45] G. Gössler and J. Sifakis, "Composition for component-based modeling," in *Sci. Comput. Program.*, vol. 55, no. 1-3, 2005, pp. 161–183.

[46] G. Abowd, R. Allen, and D. Garlan, "Formalizing style to understand descriptions of software architecture," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 4, no. 4, pp. 319–364, 1995.

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

[48] M. Proetzsch, T. Luksch, and K. Berns, "Development of complex robotic systems using the behavior-based control architecture ib2c," *Robotics and Autonomous Systems*, 2009.

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

[50] R. Ramsin and R. Paige, "Process-centered review of object oriented software development methodologies," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, pp. 1–89, 2008.

[51] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, and B. Álvarez, "V3CMM: A 3-view component meta-model for model-driven robotic software development," *Journal of Software Engineering for Robotics*, vol. 1, no. 1, pp. 3–17, 2010.

[52] H. Martínez-Barberá and D. Herrero-Pérez, "Programming multirobot applications using the ThinkingCap-II Java framework," *Advanced Engineering Informatics*, vol. 24, no. 1, pp. 62–75, 2010.

[53] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based rt-system development: OpenRTM-Aist," *Simulation, Modeling, and Programming for Autonomous Robots*, pp. 87–98, 2008.

[54] B. Dittes, M. Heracles, T. Michalke, R. Kastner, A. Gepperth, J. Fritsch, and 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.

[55] B. Dushnik and E. Miller, "Partially ordered sets," *American journal of mathematics*, vol. 63, no. 3, pp. 600–610, 1941.

[56] C. Ponce, S. Lomber, and R. Born, "Integrating motion and depth via parallel pathways," *Nature neuroscience*, vol. 11, no. 2, pp. 216–223, 2008.

[57] U. Schmidt, Q. Gao, and S. Roth, "A generative perspective on MRFs in low-level vision," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. IEEE, 2010, pp. 1751–1758.

[58] V. Willert and J. Eggert, "A stochastic dynamical system for optical flow estimation," in *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*. IEEE, 2010, pp. 711–718.

[59] AUTOSAR. (2010, May) Automotive open system architecture. [Online]. Available: http://www.autosar.org

[60] Bioloid. (2009, Nov) Bioloid robot kit. [Online]. Available: http://www.bioloid.info

[61] Khondo. (2008, Mar) Kondo KHR. [Online]. Available: http://www.kondo-robot.com/EN/

[62] S. Chernova and R. Arkin, "From deliberative to routine behaviors: a cognitively inspired action-selection mechanism for routine behavior capture," *Adaptive Behavior*, vol. 15, no. 2, p. 199, 2007.

[63] C. Heipke, H. Mayer, C. Wiedemann, and O. Jamet, "Evaluation of automatic road extraction," *International Archives of Photogrammetry and Remote Sensing*, vol. 32, no. 3 SECT 4W2, pp. 151–160, 1997.

[64] J. Fritsch, T. Michalke, A. Gepperth, S. Bone, F. Waibel, M. Kleinehagenbrock, J. Gayko, and C. Goerick, "Towards a human-like vision system for driver assistance," in *IEEE Intelligent Vehicles Symposium*, 2008.