EU FP7

AMARSi
Adaptive Modular Architectures for Rich Motor Skills

ICT-248311

D 7.3
October 2011 (18 months)

Meta-model and Software Concepts for an Adaptive Component Architecture

Authors: Arne Nordmann (CoR-Lab), Sebastian Wrede (CoR-Lab)

Due date of deliverable 1st September 2011
Actual submission date 10th November 2010
Lead Partner CoR-Lab
Revision Final
Dissemination level Public
This deliverable reports on the ongoing activities in Workpackage 7, in particular the programming model and a technology mapping for a software architecture centered around the concept of adaptive components. We present an early draft of the architectural metamodel which fuses the insights on the functional architecture with software architecture concepts. Furthermore, we elaborate on a first vertical prototype utilizing the introduced concepts in simulation (yet excluding hardware) showing a combination of skills experiment already implemented in the proposed AMARSi Compliant Control Architecture. Concluding, we provide an outlook to further co-development of the software architecture and the hardware platforms.
# Contents

1 Preface ........................................... 2

2 Approach ........................................ 3

3 Architecture Metamodel ......................... 7

4 Programming Model ............................... 10
   4.1 Component .................................... 10
      4.1.1 Component Lifecycle....................... 11
   4.2 Data-Flow .................................... 12
      4.2.1 Ports ...................................... 12
      4.2.2 Domain Types ............................... 13
   4.3 Processing Strategies ....................... 13
   4.4 Component Types ............................. 14
      4.4.1 Robot Control Interface................... 15
      4.4.2 Mappings and Transformations .......... 18
      4.4.3 Skeletons .................................. 18
      4.4.4 Adaptive Components...................... 19

5 Technology Mapping ............................. 23
   5.1 Robotics Middleware ......................... 26
      5.1.1 Scalability and Openness ............... 26
      5.1.2 Software Concepts ....................... 27
      5.1.3 Discussion ................................ 28
   5.2 Component Architecture ..................... 36
      5.2.1 Component Implementation .......... 36
      5.2.2 Domain Types ............................. 37
      5.2.3 Implementation vs. Configuration .... 38
      5.2.4 Discussion ................................ 40

6 Vertical Prototype ............................... 43
   6.1 RCIOncilla .................................... 43
   6.2 Further co-development of Hardware and Software ..... 44
   6.3 Example: Combination of Skills ............ 45

7 Summary and Next Steps ......................... 50
1 Preface

The AMARSi consortium is committed to a collaborative effort for understanding rich motor skills in humans and creating rich motor skills in robots. This ambitious goal needs combined efforts on several levels, which are distributed among partners and workpackages and, consequently, tasks and deliverables. At month 18, we reach an important intermediate stage, where we consider combinations of elementary skills from three points of view: in D.6.1 we provide a conceptual approach to facilitate and operationalize further work on the architectures, in D.7.2 we show experimental work with the existing real robotic systems and in D.7.3 we provide a technical point of view which documents our efforts to create a commonly used software architecture. All these views complement and inform each other and consequently all discuss the example of trotting and reaching towards a ball from their respective viewpoints. All three mentioned deliverables shall therefore be evaluated in connection, because they provide a more complete picture of the progress in the consortium and, while self-contained text-wise, are deeply connected content-wise.
2 Approach

Nowadays, robotic systems are mostly constructed using rigid mechanisms and actuation schemes with high stiffness designed for specific tasks. The central aim of the AMARSi project is to equip robots with the richness of biological motor behavior, hence overcoming the limited movement skills of current robots. The AMARSi consortium aims at systemic integration of insights from biology and machine learning utilizing advances in hardware and software engineering combined in a modular rich motor skills architecture to achieve this goal. Progress in AMARSi will become visible in a sequence of robot demonstrations of increasing richness, cf. [1]. Work package 7 of the AMARSi project is conducting engineering research on how to model rich motor skills in software concepts and on how to integrate higher-level learning in interaction with low-level robot control.

This deliverable focuses on (i) our methodological approach for the development of the necessary software architecture enabling the integration, composition and control of first motor skill modules, (ii) the current conceptual state of this work encoded in an initial architecture meta-model, (iii) the current implementation state of the necessary software toolchain and (iv) the description of an ongoing vertical prototyping which we recently established to more closely link the involved groups with the software engineering efforts. In the AMARSi Description of Work [1], these aspects link to Task 7.2 Meta-model and component architecture for adaptive modules and Task 7.3 Integration of high-level cognitive abilities and vision.

Following our general aim of early integration and involvement of important stakeholders, we will give first indicators on how conceptual ideas developed cooperatively with WP6 (cf. Deliverable 6.1) can be grounded in the resulting software architecture. Furthermore, we will reference where the domain analysis and the interface specifications as elaborated in Deliverable D.7.1 [2] link to the concepts introduced in this report.

The described meta-model concepts and the software toolchain will be constantly validated and incrementally refined / extended throughout the following months by applying them in the vertical prototype which was designed along the combination of skills example described in D.7.2 as well as by practically involving AMARSi partners in testing and evaluation. The latter has already been facilitated by providing the initial release of the software toolchain to the project consortium at a hands-on workshop[3] where our partners were introduced to the developed concepts, tools and programming models. For the conceptual aspects, we will continue the collaboration with partners from WP6. Consequently, one of our central aims well beyond this deliverable is to establish an architectural process linking theoretical research on motor skills with software engineering science and the hardware developments in AMARSi.

\[3\] held at EPFL Lausanne in October 2011
2 Approach

One of the main challenges in AMARSi (as in the more general domain of software engineering for experimental robotics [3]) is to link together the different research activities in an integrated way such that we can demonstrate and prove insights from biology, learning and engineering in real-world robotic experiments. Due to the fact that research in AMARSi, e.g., on the theoretical foundations of a movement control architecture as well as the hardware development for complex compliant robots is continuously progressing, we need to employ a software engineering strategy which is able to efficiently keep track with new insights and continuous change as well as ease communication about the found concepts by formalizing them quickly in formats which are accessible to the different stakeholders in the project.

Hence, WP7 aims to establish a model-driven development process [4] organizing efforts of the partners working on rather theoretical aspects, hardware and software concepts in a way which allows eased communication, early technical integration, quantitative as well as qualitative validation and later also for automation of engineering tasks to ease experimentation and scientific analysis. While many general process models are proposed in the software engineering literature, we employ and adapt existing patterns for software architecture [5] development which we think are suitable for use in a collaborative research project such as AMARSi and directly linked to our goal of facilitating a model-driven development process.

Figure 2.1: Illustration of the Model-driven Software Development approach we foresee in AMARSi [5]. Boxes represent artifacts necessary for establishing a model-driven software development process. The technology-independent architecture represents the collaborative link between WP7 and WP6 while the vertical prototype and the mock platforms link WP7 with the partners working on the robot platforms. Colored boxes indicate parts we are working on right now and which are discussed in this deliverable.
2 Approach

The proposed architectural process, cf. Figure 2.1, consists of a number of tasks related to conducting research on engineering artifacts which need to be developed or specified to achieve the outlined goals. Adapted from [5], these tasks can be mapped naturally to the development goals of WP7 in the context of AMARSi and structured in two main phases:

Elaboration  This first phase focuses on the analysis and definition of functional domain concepts leading to a technology independent architecture. Within AMARSi, the architecture blueprints developed cooperatively between partners involved in WP6 and 7, e.g., as reported in Deliverable 6.1, represent this artifact. Based on these findings and extended by software architecture concepts, a programming model can be developed and provided to scientists within the project. The design of this programming model is one of the contributions described in this deliverable. An essential need to allow for developer testing, in particular in robotics projects, is to provide robotic simulators. In AMARSi, a simulation library for the Oncilla robot was developed by the partners from EPFL which is now available in the AMARSi software toolchain to serve as an easy to use mock platform for developers. This is particularly important as robotics hardware may not readily be available to every partner, may just break frequently or if there is the need for an efficient prototyping environment.

Taking into account important non-functional properties, e.g., resource efficiency, time bounded execution of code fragments or further specifics of the target platforms or of the functional architecture, a technology mapping must be defined. This maps the programming model to concrete implementation techniques also explaining which kind of software concepts are (re-)used or need to be developed in order to actually deploy and execute the envisioned experimental robotic applications. Our initial design decisions about the technology mapping will be explained in this deliverable. To validate these decisions, a vertical prototype needs to be developed which tests our approaches in a limited but representative scenario involving all layers of the envisioned architecture. From a collaborative viewpoint we decided in the project that the combination of skills example, cf. Deliverable 7.2, shall serve as this prototype. The status of this prototype and how the programming model fits this scenario will be explained in this deliverable.

Automation  The second phase builds on the insights and artifacts from the first phase but refines these and aims to establish a model-driven development methodology. The typical initial use of architecture models is to generate code for establishing system connectivity, which actually implies to generate glue code or configuration files for a chosen technology mapping. However, beyond software architecture aspects, the real benefit of a formalized architecture metamodel is that it encodes the structural insights relevant for a software implementation of a rich motor skills architecture into a concise model representing the minimal grammar of this kind of systems. This work has been started with collaborators in WP6 and our initial take on this is part of this deliverable.

With a programming model based on the ideas of domain-specific languages (DSL-based programming model) which can be used to represent instances of AMARSi architectures, efficient qualitative and quantitative experimentation is facilitated. This may

---

2A programming / specification language dedicated to a particular problem domain and/or solution technique.
2 Approach

further be supported by a model-based architecture validation step ensuring correctness of the specified architecture instances.

While these phases were introduced sequentially, they naturally include cycles involving each other and incremental refinement. For instance, while it is not possible to build applications without a technology mapping at hand, the detailed understanding of the domain is required to choose the right technology. To solve this mutual dependency, the software architecture needs to be co-developed with an application, which currently is the combination of skills prototype. Besides working on the mock platform and the vertical prototype, the primary tasks for WP7 in the last months were to define the programming model and an initial architecture meta-model based on the recent progress on the technology-independent architecture as well as the development of an initial technology mapping to start the actual integration and experimentation. With regard to the two phases, so far, most of the time was spent on work in the elaboration phase.

The remainder of this deliverable is structured exactly along these activities. Subsequently, Section 3 reports on the initial architecture metamodel while Section 4 introduces the reader to the current state of the programming model which provides the interface for developers to realize AMARSi systems in accordance with the metamodel concepts. Section 5 then describes the current technology mapping which reports on the software toolchain for the execution and deployment of the developed systems. A current example of its application is given in Section 6 where also information about the status of the co-development of hardware and software architecture is presented. Finally, Section 7 summarizes the current state, reviews recent activities and gives a short outlook on work planned in the next months.
3 Architecture Metamodel

Subsequently, we will briefly present the current status of a meta-model for the AMARSi software architecture. A metamodel describes the structure and the identified element types of an application domain. An architectural metamodel adds to this perspective important concepts from software architecture which are required to structure, technically orchestrate and deploy complex systems whose overall complexity is well beyond the comprehension of a single developer. Concrete models that are developed and specified in terms of the metamodel elements will be representative for complete AMARSi systems.

While the architectural metamodel is related to the technology independent architecture developed in WP6, a central aim in WP7 is to further develop this into a metamodel expressed in a formal syntax with defined semantics suitable for defining a DSL-based programming model. This formalization will provide an avenue for automation, e.g., for code generation or automatic model verification.

![Diagram of the architectural metamodel](image)

**Figure 3.1:** Current state of the architectural metamodel represented as a UML class diagram. Some implementation-level aspects are not shown such as further interface classifiers or configuration elements. If no cardinality is specified at an association link a one-to-one cardinality is implied.
While the discussion on the technology-independent architecture has been and will be ongoing, the blueprint presented in Deliverable 6.1 serves as a starting point to work on formalizing the technology-independent architecture in a metamodel. We already identified a number of concepts rooted in previous AMARSi efforts (e.g., the work reported in Deliverable D.7.1) that became element types of the metamodel depicted in Figure 3.1 and which are already available for instantiation in AMARSi systems, cf. Section 6.

As the concepts which are directly related to the software architecture aspects such as Component, Port, Skeleton, Configuration, . . . are explained also as part of the subsequent Section 4, Hence, the following glossary describes the the WP7 interpretation of the domain-specific elements in a nutshell:

**Adaptive Module** An metamodel type representing a pair of an ODE and a Learner exposed as an AMARSi component type and providing ODE Status information.

**ODE** Ordinary Differential Equation (ODE) class with an object-oriented interface for internal state and update functionality which wraps specific ODE implementations.

**ODE Status** Status of an ODE system based on and with respect to the observation of the ODE’s internal state (without knowledge of training data): “converged” if rate of change close to zero, “transient” else.

**Learner** Interface for mechanisms that optimize parameterized functions on the basis of example data. This possibly includes the generation of training data by sampling methods. Note that learning rules are tightly bound to the parameterized model that is subject to adaptation. Therefore, implementation of the Learner is bound to a specific ODE or Adaptive Mapping implementation.

**Space** Combines specifications of a set of Dimension elements in a multi-dimensional vector space with update rate information for timing aspects and utility functions.

**Mapping** Maps data from one Space to another space type which may feature different representations. An exemplary mapping is an inverse kinematics mapping from cartesian to joint-angle space.

**Adaptive Mapping** A parameterized Mapping utilizing an Adaptive Module to learn the mapping function.

**Transformation** Transformation of data from one reference system to another reference system in the same space, e.g., a basis change. Conceptually, this is a restricted Mapping element.

**Adaptive Transformation** Technically similar to an Adaptive Mapping but with the same restriction as a Transformation.

**Criterion** Measure between target and observed state. Different implementations (Tracking Criterion, Reaching Criterion, Grasping Criterion, etc.) evaluate target inputs, feedback information and Module Status to compute the Component Status.
Component Status  Status of an Adaptive Component which is made visible to other components and is computed on the basis of the Criterion of this adaptive component. With respect to this criterion, status values can semantically be reached (Criterion is fulfilled), converging (Criterion is not yet fulfilled), etc.

Adaptive Component  A component type combining an Adaptive Module with a specific internal connectivity (cf. Section 4.4.3) and a Criterion which together define a particular control mechanism and a partially common interface, e.g., Component Status which is exposed for higher-level control components.

Generator  Skeleton type utilizing a specific combination of Adaptive Module and Criterion, representing an oscillatory pattern of activity at its outputs. Center, amplitude, phase and frequency of the oscillation are typical control inputs.

Tracker  Skeleton type utilizing a specific combination of Adaptive Module and Criterion, representing a tracking process. Dynamics of the tracker are partially defined by the dynamics of the target input. An example for using this kind of component is to track the position of a ball with an end effector.

Sequencer  Skeleton type utilizing a specific combination of Adaptive Module and Criterion, representing the sequential generation of control outputs with an adaptive module. Feedback inputs to the sequencer are the statuses of the controlled Adaptive Components. This allows to check if preconditions for subsequent triggered adaptive components are met.

Naturally, the above definitions and the model overview in Figure 3.1 still lack the necessary detail for the desired formalization. Still, even these initial efforts on structuring the relations among the different concepts were helpful in the discussion with WP6 partners and will lead to changes in the technology-independent architecture that again will be formalized and validated for realization in the WP7 metamodel.

We expect that once this cyclic and iterative approach is established, progress on the metamodel will speed up which is particularly important as the need for specification and formalization of these aspects is an essential precondition to conduct research at an architecture level as envisioned in AMARSi and achieve the desired automation of repetitive engineering steps.
4 Programming Model

This chapter maps the metamodel from Section 3 to a domain-specific programming model of the AMARSi Compliant Control Architecture (CCA). It will map the conceptual ideas of the metamodel to technical concepts and detail the relation between the domain-specific and the more technical parts of the software architecture.

The AMARSi Compliant Control Architecture (CCA) is an event-based component architecture for robotics research, focusing on (real-time) control of compliant hardware and enabling machine learning of rich motor skills. Components are the functional building blocks of CCA. We’re aiming for a system decomposition on a mid-level component granularity for re-usability. Connections between components in CCA are done via Data-Flow, a concept to de-couple building blocks of a system and allow concurrently executing processes.

A system or application in our architecture is a graph of loosely coupled re-usable components and the (re-configurable) connection between them. The following sections will detail the technical concepts and elements of the AMARSi Compliant Control Architecture.

4.1 Component

A CCA component is the processing element of the system, which contains the algorithmic of a sub-part of the application. In order to work together with other components, a CCA component can have an arbitrary number of input and output ports. Each port is typed (which kind of data is sent) and has a scope (semantic of the data). Input ports receive incoming data and...
provide them for computation inside the component. Components can publish their computation results over their output ports. Ports can either communicate locally with components in the same process or communicate over network with components on other machines.

Each input port has a buffer of an arbitrary (yet configurable) size to decouple computation and different data consumption rates of connected components.

The CCA component also provides a life-cycle which handles the different states of a component and the transitions between them.

### 4.1.1 Component Lifecycle

We extracted different states of a component and will continue to find necessary states. Right now, as depicted in Figure 4.2, a base component provides on top of the initial state the following component states: created, loaded, started and running.

- **Created** is the state the component is in right after creation.
- **Loaded** is the state the component is in after it was loaded from any kind of persistence. This could be just loading a (default) configuration or could be recovering from a fully persisted state of a neural network.
- **Started** is the state a component will usually pass to right after loading and will be in after it is stopped during execution.
- **Running** is the state the component will remain in until it is stopped. In this state the actual runtime processing takes place.

Between those states, the component defines the legal state transitions, as shown in Figure 4.2. In these state transitions, the CCA component will execute generic behavior and also call hooks, where the author of a component can execute component-specific code.
4 Programming Model

Examples for generic functionality executed by the component during state transitions is:

- Created $\rightarrow$ Loaded: Load a defined set of configuration parameters.
- Loaded $\rightarrow$ Created: Save the defined set of configuration parameters.
- ...

4.2 Data-Flow

A collection of components is connected to communicate with each other to form a more complex application. Within Compliant Control Architecture we go for an event-based data-flow-like communication. Data-flow being “... where a group of deterministic sequential processes are communicating through unbounded FIFO channels.” The data-flow consists of components in a graph connected by links with queues for interchanged data. The graph represents an application or program and the components represent functions to be applied to data. Data-flow links transport the data between connected functions. The data flows along the links from component to component. Components receive data from their input ports, applying their algorithmics to the data, and output the results to its output ports. Advantages of using data-flow to connect components are:

- Data-flow leads to decoupled processing
- Due to the decoupled processing use of parallel computing is inherent
- A graph is much easier to understand then pure source-code

To form an application, the graph of connected components can express an acyclic dependency graph as well as an cyclic data-flow graphs. References between components can only be expressed by their ports and port connections, not by direct references, as this would break with the desired decoupling of components.

4.2.1 Ports

Connections and data-flow between components is handled by ports. Ports are the event-driven input and output channels of CCA components. They connect components to AMARSi spaces. Port are configured it’s space of the received data (input port) or the data it sends (output port). This also specifies the domain type of the passed data items.

Output ports and input ports can be connected, if they share the same Space (so pass data of the same domain type). The deployment configuration of a port can be configured at runtime. The configuration defines the space that an output port publishes the data to and an input port listens to for incoming data. The deployment configuration defines whether the port communicates locally or over remote transport / network.
4 Programming Model

![Meta-Model of CCA Processing Strategies](image)

Figure 4.3: Meta-Model of CCA Processing Strategies

4.2.2 Domain Types

In the AMARSi Compliant Control Architecture we believe that typing the transferred data is essential to allow domain-specific treatment of the data and domain-specific applications based on that. We don’t want just send native data types (e.g. “just doubles”) without semantics, but data passed between component, so along the edges of the data-flow graph, should be typed in CCA so that it is domain-specific types (domain types) with semantics attached to them. Doing this will allow for:

1. Checks and validation
2. Domain-specific methods for data manipulation and handling

CCA domain types are linked to Spaces and AMARSi Control Spaces (ACS) defined in the metamodel (Section 3) and [6] as they contain and describe data in these spaces.

4.3 Processing Strategies

In the previous sections we mainly provided a static perspective on the software architecture. A system is yet specified through a set of components and the connectivity between them. However, specifying the runtime behavior of a software system, we need to provide a dynamic perspective and tools to configure this. In CCA we organize the execution context of components inside an application through so-called Processing Strategies. The processing strategy of a component answers the question, when the function is executed and applied to incoming data. It can be for example be based on . . .

- certain **timing**.
- **incoming events / data**.
- **both**.

A processing strategy decides when a component is processed, based on a number of possible triggers. Right now possible triggers are the global timing signal of the system or incoming data. For coordination of the processing of a rich graph of components, we already identified a set of different strategies. They are event-driven, timed and combined strategies: **Timed**, **Moderate**, **Restless**, **Coordinated**, **Port-Triggered** and **Full-speed**.
Programming Model

1. **Timed Processing**
   Processing of the component is triggered with a fixed timing interval. A typical use-case for this is a control-cycle, where even real-time may be desired. The timing is based on a global timing signal which the nodes is subscribed to. For each timed node it can be configured, how many of this global timing signals lead to a processing.

2. **Port-Triggered Processing**
   An additional configuration flag defines, which port triggers / can trigger processing of the node, if new input arrives. This is used in cases where you want to use all incoming data for processing, but processing is triggered only when certain data (read: data on a certain port) arrives.

3. **Moderate Processing**
   Processing of a node is triggered as soon as there is new data input on each and every input port of the node.

4. **Restless Processing**
   Processing of the node is triggered on every input, regardless the port.

5. **Full-speed**
   The component is executed continuously, starting the next processing cycle as soon as the current processing is finished.

6. **Coordinated**
   The coordinated processing strategy was extracted from components working with the internal model of the robot. Processing is only triggered, if all input ports have inputs from the same time of origin.

The above strategies are the base strategies we identified, but can be easily extended by framework users. The strategies allow for loose, input-driven coupling of nodes, but also for crafting control-loops with tight timing constraints.

### 4.4 Component Types

Where the base component is rather generic, during our domain analysis we identified a set of domain-specified component for the Compliant Control Architecture. These are:

1. **Resource Nodes**
   Representation of the robot platform (hardware or simulation as part of the Robot Control Interface).

2. **Mappings / Transformation**
   Components that provide mathematical transformation within AMARSi spaces or between different AMARSi spaces.

3. **Connectivity Skeletons**
   Components to ease handling of rich data-flow connectivity graphs.
4. Adaptive Components

Software representation of the AMARSi Adaptive Components (AAC).

These component types are the ones we identified so far and that will be detailed in the following sections. More component types will be identified along further discussion with our partners (especially in work-package 6) and along vertical prototypes and further experiments.

4.4.1 Robot Control Interface

*For a further description of the Robot Control Interface, see AMARSi Deliverable D.7.1.*

Components to represent the robot platform, either hardware or simulation, are collected in the component type ResourceNode, as part of the **Robot Control Interface** (RCI). Components of this type build on the extensive domain analysis we presented in our Deliverable D.7.1 [2]. Results since then were co-developed with the AMARSi platforms and will be in the continuance of the project.

The development of the unifying AMARSi robot control architecture and the corresponding API functions follows a *model-driven approach* abstracting from a concrete hardware platform while allowing to integrate or generate the necessary platform-specific code. The description
Programming Model

of the logical architecture and API views yields a first definition of the interfaces between the elements of the envisioned architecture.

Components defined within RCI represent robot hardware as well as robot in simulation and therefore are a prerequisite for early experimentation in the project both in simulation and in a subsequent stage on real robots. The collection of resource nodes form a low-level robot programming interface (API) specifically considering the requirements of compliant actuator control, proprioceptive sensing, and machine learning. In terms of the previously described data-flow graph, components of the Robot Control Interface — the ResourceNodes — are . . .

- . . . the **source** of sensor data (DTOs) of the robot
- . . . the **sink** for command data (DTOs) sent to the robot
- or both.

They represent the proprioceptive features of a robot. By *proprioception* we understand sensory modalities providing continuous feedback about the status of the body. It senses whether the body is moving and where the body parts are located in relation to each other. In case of a typical motor system this includes encoder values specifying the position of a joint. This also includes velocities, accelerations and forces applied to the robot as well as sensory for equilibrium or balance (accelerometer, gyroscope). Analogous to the understanding of proprioceptive sensing we understand proprioceptive actuation as actuators for moving body parts or changing the state of the body in any kind. Exteroception, which perceives the outside world, is not explicitly covered in this deliverable, although respected in the design process of the architecture.

The first observation as result of the Domain Analysis is that a clear separation between sensors and actuators can’t always be made. As seen in the feature models of the domain analysis [2], proprioceptive sensors and actuators share a subset of features like reporting the current state of the joint (i.e. position, force). This is valid for almost every actuator with position control, being able to report at least the current encoder value, often also the motor current et cetera. In case of the active compliant actuators that will be integrated in the upcoming version of iCub (see [7] and [8]) a single actuator provides a rich subset of features from actuators and sensors, including measurement and control of force, torque and joint position.

Body Representation

Another part of the Robot Control Interface is inspired by applications doing cartesian position control where a kinematic representation of the platform (a kinematic chain) is necessary. In case of certain control modes like gravitation compensation or cartesian force control a dynamic chain might be necessary as well. Generally speaking, on top of knowing just sensors and actuators (represented by RCI components) it might be necessary to provide an understanding of the robot’s body. For that matter we propose a component type for representation of the robot’s body, depicted in Figure 4.5.

As shown in the domain analysis of [2] the mechanical structure of robots can widely differ. To be able to model different and complex mechanical structures we propose to represent the
Programming Model

body as collection of body parts. These body parts could either be hierarchically arranged like drafted in Figure 4.5 or be in a star-like structure [9]. A body part (like an arm, leg, hand, ...) itself consists of a kinematic representation and optionally a representation of the dynamics (mass, moments of inertia, etc.) and physics (a physical model of the body part). Body parts are associated with RCI components (ResourceNode), setting the formerly loose collection of ResourceNodes into relation and allowing kinematic and dynamic solving. The grouping of nodes into body parts allows for semantic grouping and even hierarchical coupling of ResourceNodes (for example a pan-tilt unit as “eye”, being a sub-part of the “head” part).

However, in addition to setting ResourceNodes in relation and in a kinematics and dynamics context, we propose a further, optional class, providing a physical model of the robot. The body class on top is responsible for providing access to the chains and physical models.

Development of this component type will benefit from experiences from our partners, e.g. [10].

RCI Data Transfer Objects

Also part of RCI is a set of domain-specific DTOs we extracted throughout the domain analysis in [2]. A reasonable set of domain-specific DTOs for our first experimentations is already implemented. Examples are:

1. **Joint Level Proprioception**
   Joint Angles, Joint Torques, …

2. **Cartesian Positions**
   Cartesian Pose, Translation, Rotation, …

3. **Cartesian Forces**
   Forces, Torques, Wrenches, …

These DTOs provide domain-specific methods for creation and data access as well as data manipulation.
4.4.2 Mappings and Transformations

A further type of components in AMARSi applications are Mappings and Transformations, whereas Transformations are a special case of Mappings. Mappings in general have one input and one output port and map incoming DTOs from one space into another (cf. Spaces and AMARSi Control Spaces (ACS) [6]). Transformations as a special case of Mappings also have one input and output port, but in the same Space.

Examples of Mappings are Forward and Inverse Kinematics, which map from one space (joint angle space) into another (cartesian task space) and vice versa. Examples of Transformations are Translations ans Rotations in cartesian task space.

4.4.3 Skeletons

To connect components in a rich data-flow graph we on the one hand build on top of the hierarchical bus logic of RSB (see Section 5.1). On the other hand we provide a component type termed Skeletons to facilitate and ease common connectivity patterns and make them reusable across experiments and applications. The idea of skeletons goes along state of the art in parallel computing, where so called algorithmic skeletons [11] are used to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by combining the basic ones. This way even complex data-flow connectivity graphs that will probably evolve in complex applications can be glued together and simplified by re-using skeletons.

Connectivity Skeletons  We already identified and implemented a set of basic connectivity skeletons.

- **Splitter**
  A splitter (or multiplexer) splits an incoming, multi-dimensional Data Transfer Object into a set of one-dimensional Data Transfer Objects.

- **Collector**
  A collector (or de-multiplexer) fused an incoming set of one-dimensional DTOs into one out-going multi-dimensional DTO.

- **Pipeline**
  A pipeline is a common algorithmic skeleton in parallel computing, which builds a series of decoupled computing steps.

- **Farm**
  A farm is a common algorithmic skeleton in parallel computing, which distribute tasks with high computational load to a set of identical and simultaneously working computation nodes (CCA components).
Note that these skeletons are components or sets of components, that can be configured and pre-configured with control strategies. A Collector for example is pre-configured with a Moderate processing strategy, to only send the collected DTO if every input port got a new item. A Pipeline for example is a set of components pre-configured with the processing strategy Port-Triggered to work on every incoming input to compute and pass through an incoming DTO as fast as possible once it enters the pipeline.

**Complex Skeletons** More complex skeletons will be built up in AMARSi during further experimentations and along further findings in architecture (work-package 6) and along further vertical prototypes. We expect connectivity skeletons and hierarchical bus to help coping with the complexity of spaces that comes with the concepts detailed in [6]. An example of a more complex skeletons are the skeletons building CCA Adaptive Components, as detailed in the following section.

### 4.4.4 Adaptive Components

A central aspect of AMARSi and the software architecture is the Adaptive Modules / Adaptive Components, which form the main functional building blocks of AMARSi [6]. It is therefore one of the most important – because also the most domain-specific – component types we want to support in the AMARSi Compliant Control Architecture. Discussion inside the project is not yet at a stage where we can present a final blueprint of an adaptive component in software, but discussion between work-packages 6 and 7 already led to a basic idea, formulated in [6]. Sections 3.3 and 3.4. We will therefore present our current implementation of the ideas presented in those sections and propose how to proceed once the idea of AMARSi Adaptive Components is more settled.
4 Programming Model

An AMARSi Adaptive Component (AAC) is an adaptive module (e.g. an ODE) together with its inputs and outputs, a control logic and timing / life-cycle management [6]. Semantics of some of the inputs and outputs as well as the semantics of the basic control logic and internal connectivity inside the AAC is already known in the current AAC implementation. Therefore we can provide another skeleton of components and their defined scopes and connectivity, which together builds the AAC. Our implementation of an AAC additionally provides an extended life-cycle (and therefore extended life-cycle management), that adds to the basic life-cycle of the CCA base component.

Figure 4.6 shows an exemplary skeleton for the AMARSi Adaptive Component type Tracker. The skeleton defines the connectivity of an Adaptive Module, a tracking criterion that compares the target value and the actual value, and optionally mappings in between. Figure 6.3 in Section 6.3 provides an example of a concrete configuration of this skeleton in our vertical prototype. The semantics of the ports are:

1. **Reference Input**
   Reference input is the input the Tracker component needs to track.

2. **Feedback**
   The actual feedback from the robot, that can be compared to the reference input.

3. **Control Input**
   Additional control input to effect the behavior of the adaptive module.

4. **Control Variable**
   Output of the Adaptive Module, typically the command sent to the robot.

5. **ODE Status**
   Status of the ODE (in its internal state space: converged, converging, ...)

6. **Adaptive Module Status**
   Status of the Adaptive Module (is it learning (online, batch), is the command in the trained area, ...)

7. **Adaptive Component Status**
   Status of the Adaptive Component (based on 5. and 6.): Is the tracker converged, is it not converged but in the training area, is it outside the training area, ...).

Inputs and outputs (except status outputs) optionally include a mapping, to map data from the space of the input port to the space where the adaptive module can operate in. This configuration of arbitrary mapping component on these ports will later allow for a flexible usage of Adaptive Components in different spaces. For most of the Adaptive Module / ODE approaches within AMARSi, the form of data representation is vital. Mapping on the input and output allow the Adaptive Module / ODE to work in the preferred or necessary representation (space), decoupled from the actual spaces of the input or output ports (as long as there is a valid mapping between those spaces).
4 Programming Model

Figure 4.7: Deployment of Components in different Execution Environments. Components inside one environment can be connected via copy-free inprocess transport (components b, c in environment e2). Components in different environments are connected via remote transport (components a, b in environments e1, e2).

Flexible / Configurable Deployment

CCA components are executed together in one execution environment per machine. (Technically speaking, the execution environment is providing the main method.) Within one execution environment components can communicate via inprocess transport, components between components of different execution environments is done via remote transport (see Ports, Section 4.2.1).

The requirement analysis – especially in the context of AMARSi – clearly showed the need for a flexible deployment of components. Providing an architecture that integrates low-level control components as well as high-level learning and interaction components needs to be able to swap certain (computationally expensive) parts of the system to remote machines: especially Adaptive Components that can be computationally complex, but also kinematics and dynamics control modules (inverse kinematics for multiple limbs).

Computationally expensive parts of the system can be represented as a CCA component or groups of CCA components and deployed on a workstation or compute cluster. Components on this cluster will communicate in a fast local inprocess transport with each other, communicating via a remote transport with other workstations, clusters and the robot system. Since type of communication (remote, local) is configured at runtime, a CCA system (graph of CCA nodes) can be deployed in an almost arbitrary configuration across different computing devices.

A typical setup of AMARSi will be to run the ResourceNodes (see section 4.4.1) on the robot, running all other components on a powerful workstation PC, especially during development but probably also later during execution if the robot platform is not computationally capable of driving all components. Components with high computational load, e.g. inverse kinematics or dynamics for a big number of DOFs, or machine learning, can be deployed on a second workstation or even a computation cluster. The chosen technology mapping of the middleware layer (RSB) allows us to flexible deploy components in the above manner.

The former setup also holds for running experiments in simulation. In case of simulation the user can chose during configure time, if the simulation runs on the same machine as the
entire application, which would use the fast inprocess transport between simulation and the application and could be a good use-case for fast online learning. On the other hand the simulation can run on a second workstation, separating the computationally expensive physics and graphics simulation on the one hand and the application on the other hand onto two machines.
5 Technology Mapping

The concepts defined in the programming model introduced in the previous section need to be realized with a software toolchain that provides a quality of service which matches the requirements of the application domain and addresses additional non-functional requirements. As an implementation of all of the required software functionalities is naturally impossible with the allocated resources, we will explain which aspects of the software toolchain can be realized with existing open-source robotics software and adapted for AMARSi and which parts we consider critical such that they need to be developed as part of the scientific and engineering work in WP7.

This section will briefly explain some of the non-functional requirements which are important for a decision on a technology mapping, report on the overall technological architecture and explain in more detail the two layers of the software toolchain which are currently in our focus, namely the middleware and the component layer.

To define a reasonable initial technology mapping and programming model, a requirement analysis was conducted on the architecture level continuing our previous work reported as part of Deliverable 7.1. Further functional requirements are contained in the Deliverable 6.1 on the conceptual architecture. However, in contrast to these functional aspects, several non-functional requirements can be identified such as:

- Efficient on-board execution of algorithms, e.g., without threading overhead
- Components operating on different time-scales (e.g. machine learning vs. control)
- Multi-threaded processing for parallel computation
- Distributed deployment and / or autonomous operation
- Same interface (and behavior) between simulation and hardware
- Dynamic changes to component connectivity for adaptive reconfiguration
- Features facilitating experimental research on a whole systems level
- Export of AMARSi components / subsystems into other toolchains
- Import of external components into the AMARSi architecture
- Explicit (ideally executable) models of adaptive control behavior
- Sharing components, systems and data (e.g., for benchmarking)
- Use of domain-specific libraries, programming languages and operating systems
While all these requirements are reasonable, some are even conflicting such as the wish to continue using a particular domain-specific library with potentially conflicting implementation details while on the other hand components and subsystems are requested to be shared. Therefore, at each layer of the overall architecture exists a trade-off between some of the aforementioned requirements. From a high-level perspective, the AMARSi software architecture will be comprised by four distinct layers as depicted in Figure 5.1. Each layer addresses a specific set of the aforementioned requirements and is mapped to a technological solution that is either re-used and adapted or which is developed within the project:

**Domain Models** The topmost layer will mainly be the DSL-based programming model which at its core features a Domain-Specific Language (DSL) implementation. We are confident that in AMARSi (as recently demonstrated already in other advanced robotics projects [10]) a DSL will allow to express our domain-specific problems or solutions clearly and understandable for all involved disciplines in the AMARSi consortium. The DSL should i) reflect findings regarding the architecture ii) be able to express problems, solution, applications and experiments and iii) should allow to generate or configure executable software to run either in simulation or on the real robot hardware. This layer shall address requirements such as the call for executable models of rich motor skill systems and the facilitation of an efficient experimental research process. While the design of the DSL and the code generators are essential developments within the project, we will base this work on language workbenches such as JetBrains Meta Programming System or frameworks such as XText to ease DSL and generator development.

**Domain Elements** System instances described in the domain-specific language will be composed by instances of meta-model elements. These instances, referred to as domain elements, realize a particular motor skill such as a pattern generator or tracking con-

---

1See http://www.languageworkbenches.net/ar.html for a recent overview and comparison.
5 Technology Mapping

troller. Further domain elements for instance provide access to robot devices either in simulation or on the actual robot platform / hardware, are ODE implementations encapsulated in an adaptive module or are just transformation modules mapping a data from one space into another. The components in this layer mostly realize functional aspects. At the time of writing already some domain elements use the proposed programming model, e.g., the Oncilla simulator.

Domain-specific Architecture Domain elements are executed in a domain-specific software architecture implementing the AMARSi programming model and further functions either required by the architecture metamodel presented in Section 3 or due to the execution environment. The so-called AMARSi Compliant Control Architecture (CCA) is a light-weight component architecture implemented as a C/C++ library as a part of our WP7 tasks. It provides the necessary implementation logic to assemble component-based AMARSi systems. Exemplary aspects addressed in this layer are efficient periodic execution of components operating at different timescales and event-driven components, their lifecycle management and composition. This composition process will be automated once the DSL-based programming model is in place. Domain-specific components are provided by and shared between the AMARSi partners. This addresses the desired code re-use and eases the setup of robotics experiments.

Middleware While the domain-specific functionalities are implemented inside CCA, many of the requirements outlined before can typically be delegated to a (robotics) middleware. For instance, the ability to distribute or co-locate components in the same process should be supported by a middleware framework and not be part of the domain-specific architecture. As the development of a new middleware is out of scope of AMARSi, we currently decided to adopt an event-based middleware (termed RSB [12]) recently developed and used in another FP7 EU robotics project due to its small footprint, extensive configurability and openness. Additionally, it provides the required tools for experimental research and integration with other relevant frameworks such as ROS [13] or YARP [14]. Additionally, the middleware level allows the technical integration of high-level components implementing cognitive models outside of the core AMARSi scope such as computer vision components. By employing a middleware approach at the lowest level, the sharing of experimental data and loosely-coupled integration of so-called unmanaged components becomes feasible.

In the remainder of this section we will focus on the two bottom layers: the domain-specific software architecture realizing the programming model provided to AMARSi partners and the currently chosen middleware as an important part of the technology mapping. These layers are already implemented or integrated in the software toolchain. Initial implementations of domain elements in the third layer were recently implemented and are being tested in the vertical prototype explained in Section 6. Implementation of the fourth layer will be started soon as the initial architecture metamodel is evolving as explained in Section 3. That said, the evaluation of the necessary software tools for this layer has already been started.

2 Unmanaged implies here that the execution thread of these components is not controlled by the architecture as is the case for CCA components.
5 Technology Mapping

5.1 Robotics Middleware

The primary technology employed at the middleware layer in the AMARSi Compliant Control Architecture utilizes an open-source middleware termed Robotics Service Bus (RSB) [12]. This framework which is similar but nevertheless conceptually and technically different from popular robotics middleware approaches such as ROS [13], YARP [15] or OpenRTM [16] is in our opinion well suited as a basis for this architectural layer.

Due to the fact that this approach originates from the same institute as the AMARSi software architecture, requirements of AMARSi are considered first class in its development roadmap. After a short explanation about the requirements that lead to its development (which are very much inline with the AMARSi needs) and a brief introduction into its core concepts, a comparison based on qualitative discussion and quantitative benchmarks will further underline why we favor this approach over, e.g., ROS or YARP.

5.1.1 Scalability and Openness

In the introduction to the technology mapping, we outlined already very briefly a number of requirements which need to be considered by robotics middleware technology. Two requirements which are of special importance for AMARSi and which were particularly considered when RSB was developed are openness and scalability.

In the context of distributed software systems, openness is the property that services provided by a system adhere to standardized protocols which formalize their syntax and semantics [17]. According to this definition, openness of a system leads to the following desirable qualities:

- **Portability** The property of being able to function in different execution environments without modification. Examples include different target platforms and their respective constraints regarding available (parallel) processing, memory and network resources. Hence, aspects like dependencies or the use of threads may be severely restricted.

- **Flexibility** The ease with which the structure of a software system can be changed, e.g., by adding new components or altering behaviors of system parts.

- **Interoperability** The ability to function in conjunction with other systems designed for the same domain. Components written for one middleware shall ideally be directly usable in another middleware if both have sufficient interoperability qualities.

In addition to openness, we consider the following aspects of scalability important (derived from [17]) in the context of AMARSi:

- **System size and distribution** the system size and type which can be handled by a middleware. E.g. components in a single process, multiple processes on a single node or on distributed nodes in a network. This includes the number of components as well as their platform and runtime context. Scalable systems foster the integration of components. The ability to scale the size of a system implies sufficient efficiency in its processing.
5 Technology Mapping

- **Organization** Ease of developing / maintaining a system if it is developed by several organizational structures with potentially overlapping / conflicting aims and guidelines.

While for the design of RSB more requirements were considered, these two aspects are critically shared with the AMARSi project. Consider the RoBoard 110 embedded board of the Oncilla robot as an exemplary execution environment for control components in AMARSi. Despite other positive attributes, it provides only very limited computing resources and thus defines the lower end of the performance spectrum while on the other end partners probably want to use a powerful computing grid as a backend for more complex learning processes within the same software architecture.

### 5.1.2 Software Concepts

In a nutshell, RSB is a message-oriented, event-driven middleware aiming at scalable integration of robotics systems in diverse environments. Being fundamentally a bus architecture, RSB structures heterogeneous systems of service providers and consumers using broadcast communication with $m:n$ semantics over a hierarchy of logically unified channels instead of a large number of point-to-point connections. Nevertheless RSB comes with a collection of communication patterns and other tools for structuring communication, but does not require a particular functional architecture or decomposition style.

RSB is implemented as a flexible, lightweight toolkit. Based on previous experiences, the design and implementation try to avoid centralization and complicated dependency structures. Instead, functionality is organized in layered packages with minimal, clearly stated dependencies and well defined extension points as shown in Figure 5.2. The core library merely supplies the abstractions and machinery for building communication systems subject to varying requirements.

---

[3]The reason for the similar motivation in HUMAVIPS is that the middleware needed to comply with equally limited resources available on the embedded board of the NAO robot manufactured by Aldebaran Robotics.

[4]RSB is available as open-source software at: https://code.cor-lab.de/projects/rsb

---

![Figure 5.2: Core API elements and extension points in the RSB architecture.](image-url)
5 Technology Mapping

RSB does not include a custom IDL for data description. Instead, an additional project termed RST provides common data definitions using an external IDL (Google Protocol Buffers) which is used for generating efficient de-/serialization code. This IDL is well-known outside the robotics community and is not inherently coupled to a broader framework, nor is the RST project itself limited to RSB. RSB provides particular support for this kind of serialization format but does not require its application.

This core is implemented in multiple programming languages (currently C++, Java, Python and Common Lisp) with the aim to provide a natural interface in each language. RSB is not tied to a particular event dispatching strategy and threading strategy, network transport, serialization mechanism or programming language. Following this principle, dependency footprints of the core layer implementations are as small as possible and Linux, Mac and Windows are supported. For instance, the RSB core (for the C++ implementation) solely relies on the Boost [18] library providing a basic but fast transport based on TCP sockets. Moreover, the (optional) Spread-based connector requires the Spread Toolkit, which is an easy to compile C-library for reliable multicast communication, and Google Protocol Buffers for internal serialization of event notifications. This small footprint allowed for instance the application of RSB on NAO without an otherwise much more complicated cross-compilation setup.

A first example for the utility of these extension points in AMARSi is the implementation of a single-threading event dispatching strategy without any locking or queuing which is under development for use on the RoBoard 110 to avoid the complexity of context switches due to multi-threading. While RSB itself is not a hard real-time approach, we aim to make it usable in real-time tasks. This will be further explored also in the vertical prototyping once the dedicated test setup for the Oncilla hardware is available (cf. Section 6).

These conceptual and implementation properties shall allow RSB to scale across a wider range of diverse functional requirements, heterogeneous hardware platforms and varying reliability and timing requirements than other robotics middlewares. Additionally, RSB’s open architecture and project structure and lightweight implementation enable its use with small entry barriers and little framework lock-in. Furthermore, much effort is put into systematic testing and continuous integration.

For more information on the implementation concepts and the available toolchain for RSB, we refer the interested reader to [12] and the information available on the project website.

5.1.3 Discussion

Subsequently, we will briefly discuss related robotics middleware approaches. For our comparison we selected ROS as a representative of the state-of-the-art in robotics system integration demonstrated by its wide adoption and YARP as it has been the robotics middleware used in the RobotCup EU project which is relevant for the work done in AMARSi.

The first part of this discussion is qualitative which may be subject to a strong bias on the introduced requirements while the second part is quantitative and includes initial benchmarks on the Oncilla embedded PC platform.
5 Technology Mapping

ROS Communication System Willow Garage, the makers of ROS (Robot Operating System), aim to provide an open-source, meta-operating system supporting robotics software development in different robotics domains with a focus on mobile manipulation. Over the last years, ROS gained wide community support. It provides access to a large number of software libraries for building robotics systems, which expose their external interface using features of the ROS communication stack called ros_comm. Due to the wider focus of ROS, our approach mainly compares to this communication subsystem.

The ros_comm stack implements a type-based, anonymous publish/subscribe model where each logical connection between a set of publishers and subscribers is bound to a symbolically identified Topic which prescribes the type of exchanged data with this channel. Conceptual differences to the concepts available in RSB are the lack of hierarchically organized topic identifiers and the restriction to a single data type per topic whereas RSB allows polymorphic channels. Polymorphic channels remove the necessity to couple namespaces with data type design, which facilitates the development of generic software components such as the RSB/ROS bridge developed for integrating ROS components in RSB systems and vice versa. While ROS also provides content-based filtering with the message_filters package, installed filters are not visible to the transport layer and hence cannot be optimized like as it is the case in RSB.

Besides publish/subscribe, ROS supports Services which offer remote procedure calls. In contrast to RSB, ROS services are conceptually outside of the channel system described above. In RSB an RPC-like request/reply pattern is logically implemented on top of the unified event bus. This facilitates easier implementation of advanced event-based RPC patterns with optional feedback such as the Task-State pattern and allows us to apply the same toolchain, e.g., for introspection or recording. Performance drawbacks caused by this implementation can be mitigated if necessary by using special connectors which implement RPC interaction natively at the network-level. However, in contrast to ROS and other approaches, this optimization is not visible to client code or even framework tools.

The ROS communication stack may very well scale to larger distributed robotics systems. However, it remains unclear how it is able to support more tightly integrate component interactions. To this end, no direct equivalent to the presented in-process transport for collocated optimization is available. The comparable features we could identify, in particular Nodelets and intra-process publishing seem to require changes at the level of the client code which would prevent seamless reconfiguration of ROS components to scale down to embedded platform and integration of components in more tightly coupled feedback controllers. The RSB architecture allows this kind of configuration change without code-level changes.

While the aforementioned differences influence (among other aspects) the scalability of a system integration solution, one of the primary goals of the ROS communication library as well as RSB is to be as lightweight as possible. Here, both frameworks are on an equal footing with the Boost libraries as their primary dependency and additions for communication with the

---

6. ROS allows for hierarchical syntax in topic identifiers but does not enforce or associate any semantics with it. Redirections into a common topic or sub-topic would require developers to know beforehand every possible topic to be aggregated.
5 Technology Mapping

ROS Master (XML-RPC) in ROS or Protocol Buffers as the internal serialization mechanism in RSB. However, ROS lacks clearly defined extension points for important functionality, e.g., for 3rd party serialization formats or different transports.

**YARP**

YARP is a robotics middleware which has been used for the system integration in a number of projects on advanced robots such as Kismet, COG or currently in the context of the iCub humanoid robot. With our it shares approach the motivation to not become an operating system for robots but to focus on the middleware aspects of robotics system integration. Besides its dependency on ACE, YARP is a lightweight framework which allows the execution on platforms with limited processing capabilities. Openness is additionally supported by a number of available Carriers which realize different transports such as TCP, UDP or also local transport.

Using YARP, an integrated system is organized as a number of independent components which expose their interface via Ports that allow to send data to other participants. Port communication implements a distributed observer pattern, which is similar to RSB’s event bus concept. However, compared to its implicit invocation architecture, ports are directly connected to each other through an explicit connection which has to be established manually. While a hierarchical name scheme typically appears in port identifiers, e.g., in the iCub interface, their syntax is rather unspecified and the hierarchical structures do not impose any semantics.

The default serialization in YARP is the Bottle format. At client-level, the bottle concept provides a simple interface where native data types, strings, lists and dictionaries can be added and removed to a bottle instance. While this represents a straightforward concept, the lock-in at code-level would be very high if used extensively. Furthermore, no IDL for specifying data types and no generators for encoding or decoding code exist. That said, YARP supports a limited number of predefined data types natively, e.g., OpenCV images, and users may write their own serialization engines. Besides that, content-based filtering is not directly supported neither at client-level nor within the framework.

**Quantitative Comparison**

The aim of the subsequent benchmarking experiments is to to quantitatively validate that the performance of RSB (without performance optimization) is comparable and sufficient for the AMARSi use cases. While the former aspect can be evaluated as only the absolute performance measures are compared, the assessment of the latter may need more insight into the required performance which we will gain, e.g., by further extending the benchmarking environment towards a more realistic hardware and software environment with comparable computational load. Nevertheless, already the comparison against other middleware provides an initial technical validation of the chosen technology.

Hence, we performed a benchmark of the C++ implementations of RSB, ROS and YARP in a single scenario but in different deployment configurations. For this purpose, the different middlewares were compared against each other in a round-trip experiment. A sending component,
the driver sent a ping message to a varying number (fan out) of replying components, the so-called reflector components, which replied with a pong. The round-trip latency between sending the ping message and receiving the pong message was measured for different message sizes. We required reliable communication, disregarding, e.g., the sending of unsafe message in the RSB spread transport.

The receivers where started in a single process (i.e. multiple listeners for RSB) and communicated with the sending component, which was launched in a separate process in different configurations involving a regular workstation and the RoBoard 110. The workstation was an Intel® Core™2 CPU with 2.40 GHz and 4 GB of RAM. The three examined target configurations motivated through the AMARSi use cases were as follows:

**RB110-only** In this configuration both the driver and the reflector were linked into a single executable process to evaluate the co-located in-process communication performance of the different middlewares on the resource-constrained RoBoard 110 platform.

**Workstation-only** This is the same configuration as above deployed on a dedicated workstation.

**RB110/Workstation** In this configuration the driver component runs on the RoBoard 110 while the reflector components run on the workstation.

Using these three different deployment configurations, the initial benchmarking evaluates framework performance along the described round-trip experiment in two typical configurations.

**Scenario A: Remote Communication** This scenario evaluates the basic middleware performance in terms of communication latency either by performing network communication between the RoBoard 110 and a workstation or between separate processes on a cluster.

Figure 5.3 depicts the mean round-trip latencies for remotely communicating processes in the RB110/Workstation configuration. This scenario mirrors a typical deployment scenario for experimentation where the robot is controlled remotely and the main processing is done on a workstation or cluster. Here the RoBoard 110 was connected to the workstation using a 100MBit ethernet connection.

The benchmark was performed with YARP, RSB with Spread and Socket transport as well as ROS. The main result of this figure is that ROS and RSB with Socket transport (RSB-SO) are slightly faster than YARP for larger message sizes, which again is slightly faster than RSB with Spread transport. RSB and ROS operate in the same order of magnitude, which demonstrates sufficient performance for RSB. Please note, that the benchmark displays an anomaly for these two middlewares for smaller message sizes which we suppose is due to an unset `TCP_NODELAY` socket option. We will further investigate this as part of the ongoing benchmarking activity.

---

9 Required services such as the ROS / YARP nameservers or the Spread daemon were deployed on the workstation.
5 Technology Mapping

The benchmark results displayed in Figure 5.4 confirm qualitatively the findings of the experiments on the embedded board. RSB-SO seems slightly faster than ROS except for the anomalies due to the suspected problem explained above.

Figure 5.3: Mean round-trip publish-/subscribe latency for RSB with Spread-based (RSB) and Socket (RSB-SO) transport plugins, YARP with TCP carrier and ROS with default TCP transport with differing fan-out and payload sizes in the RB110/Workstation setup. Latencies are displayed in logarithmic scale.

Figure 5.4: Mean round-trip publish-/subscribe latencies for RSB with Spread-based (RSB) and Socket (RSB-SO) transport plugins vs. YARP with TCP carrier and ROS with default TCP transport in the Workstation-only setup.
5 Technology Mapping

Scenario B: In-Process Communication  This scenario evaluates the middleware performance using in-process communication features. This test case is critical for the component architecture if the middleware functions are used to provide connectivity between components.

Figure 5.5: Mean round-trip publish/subscribe latencies for RSB with in-process transport plugin and YARP with local carrier in the RB110-only setup. Latencies are displayed in logarithmic scale.

Figure 5.6: Mean round-trip publish/subscribe latencies for RSB with in-process transport plugin vs. YARP with local carrier in the Workstation-only setup. Latencies are displayed in logarithmic scale.
Both on the embedded board, cf. Figure 5.5 as well as on the workstation, cf. Figure 5.6, RSB displays the best performance which can be explained as in the local case events are not copied but dispatched to listeners as immutable references. While YARP should support a similar kind of zero-copy mechanism and we read the documentation exactly, we could not induce a related effect in the benchmarking. We are going to ask a YARP developers for a code review. However, even for small payload sizes RSB is significantly faster in both configurations. While we wanted to test ROS in this setup as well using the Nodelet-API, this was not possible as at the time of writing the nodelet API was inaccessible. In further iterations of the benchmark we may compare this with the intra-process publishing features of the ROS Communication system.

**Variance in both Scenarios** While the mean latency is relevant for assessing the overall speed and throughput of a middleware, the variance in the individual round-trip measurements is of great importance for control applications. For clarity of presentation, Figures 5.7 and 5.8 depict the mean and variance for in-process in the RoBoard 110 configuration and in the remote communication condition with separate diagrams.

![Figure 5.7: Mean and variance of in-process communication latencies for selected fan-outs using RSB and YARP. This benchmark was executed in the RB110-only setup.](image)

For the in-process condition on the RoBoard the variance for RSB communication is significantly lower than for YARP as displayed in Figure 5.7. This may be linked to the zero-copy issues described above and potential dynamic memory allocation. However, in combination with object pooling for creating the event payloads and a single-thread event dispatching strategy we suppose that in the case of RSB it will even be possible to dispatch events with deterministic timing if the number of listeners does not change dynamically. This may become important for applying RSB in a real-time context such as on the Onci Ila robot. We will validate this once the hardware is integrated into the vertical prototype (cf. Section 6).
Figure 5.8: Mean and variance of remote communication latencies for selected fan-outs using RSB and YARP. This benchmark was executed in the RB110/workstation setup.

Figure 5.8 again qualitatively confirms the findings with the already described anomaly due to the TCP_NODELAY problem. For larger payloads the variance in RSB remote communication is lower than in YARP.

**Conclusion and Further Goals**  During development of the middleware and in order to continuously assess the technology in comparison and in relevant experimental conditions, we experienced the dedicated benchmarking framework as an extremely useful tool. An example for this is the TCP_NODELAY anomaly which seems to be a regression we did not experience with earlier RSB versions. The whole benchmarking code is publicly available and shall be send to the corresponding framework developers for review of “their” code although we tried to carefully implement the test drivers along the respective documentation.

Based on the current results and aside from its functional properties, the quantitative benchmark so far confirms that the chosen middleware is at least on par with its strongest competitors if not ahead of them. It outperforms its contenders particularly with regard to in-process communication which is relevant for the component architecture. That said, it will be important to include more realistic benchmark conditions such as a periodic test which measures the Jitter induced by the different framework approaches. Further tests on other aspects such as the time required to setup connections may also be relevant for an AMARSi middleware, e.g., to allow fast dynamic reconfiguration of system connectivity. Finally, while we started to work on a target execution environment, the RoBoard 110, we will need to add further components of the real robot to be able to do serious testing which will allow to assess the suitability of the middleware to run on the real robot.

[10] https://code.cor-lab.de/projects/rsbench

35
5 Technology Mapping

5.2 Component Architecture

The AMARSi Compliant Control Architecture implements component-level functionality building on top of the middleware layer and is implemented as open-source C++ library. C++ as object-oriented general-purpose programming language, that combines both high-level and low-level language features to support low-level and high-level functionality we need in complex robotics systems. CCA uses apart from the previous mentioned libraries (RSB, RST) the C++ open-source library boost and the C++ open-source library NemoMath, a header-only template library for mathematical computations in the context of robotics and learning that wraps and extends the popular libeigen and optimizes it for shared-ownership in object-oriented applications. CCA then is a first prototype technology mapping of the programming model depicted in Section 4.1 following concepts of component-based software engineering.

5.2.1 Component Implementation

The base component of CCA is implemented as abstract C++ base class that is extended to implement more specific components. The life-cycle of the CCA base component (cf. Section 4.1.1) is implemented as lightweight state machine inside the base component, that has to be extended by inheriting specific component, if necessary. The component keeps its current state and state transitions can be triggered from the component itself or from outside. Between the already identified states described in Section 4.1.1 the component defines the legal state transitions. In these state transitions, the CCA component executes generic behavior and calls method hooks, where the author of a component can execute component-specific user-code for each state transition. Every state transition calls a hook method, that a user can implement when deriving a specific component from the CCA base component. E.g. the hook onLoad() is called after the generic loading of the component is executed, to allow additional component-specific functionality.

Input and output ports of the CCA component are mapped to RSB listeners (input port) and RSB informers (output port).

Processing Strategies are implemented as abstract C++ base class and derived classes for the processing strategies we already identified. A processing strategy implementation of the base class has to specify if it relies on incoming inputs, on the timing signal or on both.

If a processing strategy defines to rely on the timing signal, it is requested on an incoming timing to decide, if the component is processed or not. This way a processing strategy can either rely completely and solely on timing, specifying after how many timing signals the component is processed, or can use the timing signal for a time-out, if relying also on other clues.

If a processing strategy defines to rely on input data of the component, it is requested by incoming events of the middleware layer (incoming data at the component’s input ports) to decide whether processing of the component is triggered. For this it gets a reference to the data of all input ports to decide based on these. Since the processing strategy gets a reference to
5 Technology Mapping

all input data, it can trigger processing based on how many DTOs on which ports are available or on the actual content of the DTOs.

If a processing strategy is requested, either through a timing signal, incoming input or both, it returns true or false. If it returns true, processing of the component is executed, including calling another method hook onProcess(), where the actual user-code of the component goes.

Timing of component and the entire application is implemented prototypically as global timing signal within one application. For this we implemented a global timing signal which we call Global Beat, where components can subscribe to. Every component with a processing strategy depending on a timing signal, needs to be registered at the global beat. The global beat is implemented as a precise periodic thread, sending a timing signal at a specified cycle time to all registered components. The timing signal is implemented using the boost-signal implementation.

Since all components are not based on real time, but on a timing signal, the entire application can be slowed down and sped-up by varying the cycle time of the global beat. This comes handy for debugging and slowing down experiments. Note that in order to also retain functionality of an application where components are involved, whose computation (not only processing strategy) depends on timing (e.g. oscillators), also the computation needs to depend on the global timing signal. This is not yet fully implemented and will be further developed also along the idea of introduce a virtual time inside CCA also to synchronize CCA applications with the virtual time inside simulators.

5.2.2 Domain Types

Data Transfer Objects (DTO) in CCA are implemented in form of a C++ base class, providing the basic (yet not domain-specific) methods for creating and accessing the contained data. Inheriting classes then add specific getters (e.g. xyz getters if representing cartesian values) and/or specific methods (like normalizing joint angles to the range \([0, 2\pi]\)). It even allows checking for hardware-specific limits (e.g. joint limits), if a DTO is platform-specific.

Immutability of all DTOs objects is essential since the data-flow semantics in the way implemented in CCA allows several input ports connected to one output port and therefore several components working with the same data. To avoid side-effects, incoming data can just be read, not manipulated, and is therefore immutable.

To represent open the data transferred in CCA for other frameworks and allow integration even on this level, we use another open-source project: Robot System Types (RST). RST is a collection of open type specifications for Robotics and Cognitive Systems, specified in Google Protocol Buffer’s IDL\(^{11}\) format. All DTOs are mapped to RST data-types to be able

---

\(^{11}\)An interface description language (IDL for short) is a specification language used to describe a software component’s interface. IDLs describe an interface in a language-neutral way, enabling communication between software components that do not share a language – for example, between components written in C++ and components written in Java.
5 Technology Mapping

Figure 5.9: Example: An RCI DTO (cartesian rotation) represented in RST and in the Robot Control Interface. The RST rotation class is generated from an IDL and is a pure data-holder with getters and setters. The according RCI DTO has a semantic understanding of contained data and provides domain-specific methods. Converters transform the DTO into the RST type for fast and language-independent remote transfer.

(a) Implemented Component  
(b) Configurable Component

Figure 5.10: Example component: a) Implemented and re-usable but not executable yet. b) Configured for a specific application context and executable.

to represent them in the IDL-specified format for openness and interchangeability. Note that domain-specific checks, validations and methods get lost when mapped to RST types and are thus only available within CCA.

In order to allow on the one hand fast data-flow in local setups and on the other hand also support data-flow in remote setups, DTOs are treated differently in local and remote communication. In case of local setups (communication between CCA components on the same machine), DTOs are passed by reference (C++ shared pointers) and thus prevent copying for fast communication. In case of a remote transport, we use a mapping of DTOs to RST data-types and therefore use the Google Protocol Buffer serialization / de-serialization logic for remote communication. Note that in this case, only the raw data (native data types, like integers, floats, strings) are passed as payload over the transport, but on both sides can be accessed as DTOs with the domain-specific methods when used in CCA. (see Figure 5.9) All data sent in this way can also be used in arbitrary third-party code (in Java, C++ or Python) when using the Google Protocol Buffer library.

5.2.3 Implementation vs. Configuration

In the above sections we described both, implementation of components based on the C++ base component as well as configuration of components and the connectivity between. Our plan with CCA is to create new applications and experiments by reconfiguration rather then
5 Technology Mapping

implementation. This will i) set the focus to re-usability of components through-out project partners and ii) will allow fast and flexible experimentation even for project partners with expert knowledge in their disciplines (biology, machine learning) but not necessarily expert knowledge in software development and iii) will allow reconfiguration of applications and experiments without recompiling and iv) – important for the AMARSi domai – allow learning on system level.

The differentiation between implementation and (re-)configuration is as follows:

Implementation / Compile-time Implementation is focused on the implementation of the actual components. Steps are

1. Creating the component
   Creating your component by deriving from the abstract C++ base component class or a more specific implementation.

2. Defining ports
   Defining input and output ports of your component. This includes specifying the data-types of each port.

3. Implement algorithm
   Implement the main processing function (the onProcess() -Hook) with the actual algorithm. This can use other libraries, but must not have any references to other components, but just work on incoming data and its own state.

4. Implement further hooks (optional)
   Implement further hooks like onStart(), onStop(), onLoad(), ....

5. Implement defaults (optional)
   Implement defaults for configuration (e.g. set a default processing strategy).

Following these steps will lead to a re-usable and managed CCA component. The component is not yet executable since configuration is missing.

Configuration / Run-time Configuring components will set them in the context of an application / experiment and prepares them for execution. In order to do so the following has to be configured:

- Instantiate components
  Choose a set of components to use and instantiate them.

- Configure strategies
  Configure your component’s processing strategies.

- Configure deployment
  Specify deployment by configuring input and output ports. This includes specifying the scope of each port, and if its a remote or local port (or both). For input ports also the input buffer can be configured (queue size, keeping latest).
5 Technology Mapping

- **Configure parameters (optional)**
  Configure any additional configuration parameters (e.g., controller parameters, learning rates, . . .) if provided by the component.

Some of the configuration steps may be left out, if defaults are given by the implementation (e.g. default processing strategies). The configuration steps will be supporting by appropriate software tools, like graphical user interface (cf. Matlab Simulink, . . .) and especially a domain-specific language.

Configuration can be changed during runtime, e.g. changing processing strategies, component parameters or connectivity between components.

5.2.4 Discussion

Subsequently, we will briefly relate our solutions to related approaches of robotics component architectures.

**ROS** Within ROS *(Robot Operating System)* Willow Garage provides component-level abstractions within a stack called *ros_node*. The *ros_node* stack implements provides the *node* abstraction, which is “a process that performs computation. Nodes are combined together into a graph and communicate with one another”. With nodes ROS aims at providing component abstractions on a “fine-grained scale”, which seems to be coarser than the “mid-level granularity” we aim at with CCA (given examples are: nodes controlling the laser range-finder, the robot’s wheel motors, localization, path planning, or a graphical view of the system).

Different types of nodes are in ROS defined as different node implementations, which can then be instantiated under different names. Groups of node types of a similar purpose – like component types in CCA – are grouped through so called *stacks*. To organize nodes in a system, nodes can be assigned hierarchical names and even a set of nodes can be assigned to a global prefix to separate nodes of different applications. Using a fast inprocess, zero-copy transport between multiple algorithms, *ROS nodelets* are used, which provides dynamic loading of classes into the same node. Using either remote or inprocess transport by just reconfiguring nodes doesn’t seem to be possible.

**YARP** YARP *(Yet Another Robot Platform* from the RobotCub project) provides abstractions on a middleware-level (see Sections 5.1.3), but does not provide any component-level abstractions. Ports can have same port-name prefixes to indicate togetherness, but this doesn’t have any functional or technical implications. Management of execution is up to the user, who can assign functionality to port callbacks or use a *RateThread* to have execution with a fixed timing. Data-types are not supported within YARP, but can be implemented by users using YARP *Bottles*, collections of native data types (string, integer, float, . . .).
Orocos

Orocos is a C++ framework for component-based robot control software with a focus on real-time motor control. The Orocos component model provides C++ class implementations for “i) lock-free, thread-safe, inter-thread function calls ii) communication between hard real-time and non real-time threads iii) deterministic execution time during communication for the higher priority thread. iv) synchronous and asynchronous communication between components and v) component distribution”.

The base component, called TaskContext provides various C++ function hooks in which the user can place custom C++ code. Component are accessible over a network and configurable using XML files. The framework also manages the setup, distribution and building of the so-called real-time software components. It takes care of the real-time communication and execution of software components and is therefore sometimes referred to as being a middleware. A special DeploymentComponent loads, connects and configures a given list of components. It can also load an XML file which describes which components to load, connect and start in the current application. Additionally provides an Orocos Device Interface which defines how to interact with analog and digital IO and encoders.

Ports in Orocos are connected using Data-Flow as in CCA, but other then in CCA data-flow is that each port can only have one connection (1 : 1) (m : n in CCA). Although the component model of CCA and Orocos are quite similar, Orocos’ choice of middleware counterdicts requirements we have in AMARSi. The underlying framework (middleware) of Orocos is CORBA to allow connection and communication of Orocos components over a network or between two processes on the same computer. CORBA is a critical dependency which discourages the use of OROCOS. At the same time it seems a rather challenging task to replace the middleware implementation of OROCOS with a more lightweight solution.

For our future work to provide real-time capable components (or a subset of components – certain component types) we will closely monitor the OROCOS efforts and check if it can be adapted for use in AMARSi. Furthermore, we plan to include it in the quantitative benchmarking at the component level.

![Figure 5.11: Creation of a Orocos Component using the Orocos Simulink Toolbox.](image-url)
Simulink

Simulink is a popular software tool in prototyping and engineering of dynamical systems. It supports creation of networks of generic and custom building blocks and simulation of the system. To be run on robots for example, Simulink supports generation of Matlab or C code from a configured system. However, Simulink is a rather general solution and doesn’t not provide too much support for problems in our domain. Additionally Matlab and Simulink are closed-source, proprietary tools and induce a rather strong lock-in.

A more domain-specific usage of Simulink is provided by a connection between Orocos and Simulink which is provided by the Flander’s Mechatronics Technology Centre as an Orocos Simulink Toolbox. The toolbox let you create and generate Orocos components from a Simulink model. The toolbox works for Simulink in Linux and Windows. A component generated with this toolbox can be compiled and loaded in an Orocos application, without writing additional code.
6 Vertical Prototype

Our development of a vertical prototype went along the implementation of a platform drivers for the AMARSi quadruped robot Oncilla and an application with combined skills. Since the hardware was delayed and is not yet available for implementing drivers, our prototyping was done in simulation. The following sections will present the drivers for simulation and an outlook on how this will translated to the actual hardware in the near future. Section 6.3 will explain the CCA implementation of the Combination of Skills experiment in simulation, which we already presented to our partners at the Oncilla Workshop in October 2011.

6.1 RCIOncilla

Porting the AMARSi Compliant Control Architecture to the Oncilla simulation (and later to the hardware) requires implementing the Robot Control Interface concept and components to the quadruped robot platform and implementing a library to use it. The library was implemented in communication with EPFL and implements RCI components in the libRCIOncilla, as already prepared on a conceptual level in [2].

In order to glue an CCA application together with a robot platform, the platform (simulation or hardware) needs to be represented by according ResourceNodes, that can be used in a CCA data-flow graph. The basic ResourceNodes for the quadruped robot were already designed in [2] and prototypically implemented and refined in the last months. For having a first vertical prototype we focussed on joint angle control and therefore the two ResourceNodes QuadHipNode representing the hip actuator/sensor of the quadruped robot and QuadKneeNode representing the knee actuator/sensor. Figure 6.1 shows how the two resource nodes are embedded in a CCA subgraph together with further nodes in order to use the simulated quadruped robot within CCA.

In the Webots quadruped simulator, the QuadHipNodes and the QuadKneeNodes are the RCI ResourceNodes representing the robot. They exist four times each in the Simulator to represent all four legs of the robot, the three duplicates of each of the Resource Nodes are left out in Figure 6.1 for layout reasons. The WebotsSynchronizer node synchronizes the ResourceNodes with the simulator (commands send to the ResourceNodes are set on the simulated actuators, simulated sensor values are read and set in the ResourceNodes) in a specified timing interval. Figure 6.1 shows the configuration of the simulator setup for the remote case(application running on a second machine). Although eight joint angle values (four hip values, four knee values) need to be send to the robot, we instantiated a Splitter component, that receives one eight-dimensional joint angle DTO and splits it up to send the eight one-dimensional joint...
angles to the eight ResourceNodes via inprocess transport. This way we reduce the amount of data items sent over network by factor eight. Sending the joint angle sensor values over network is done accordingly using a Collector component.

Important is, that the exact same setup will be used in case of the real robot hardware. In this case the WebotsSynchronizer now synchronizing the ResourceNodes with the Webots simulator, will then be exchanged by the hardware drivers for the quadruped Oncilla robot in liboncilla. This will expose the same interface and usage independent from using the simulator or the real robot.

The simulator was already tested and used for CCA experiments at the Hands-On Oncilla Workshop in October 2011 in Lausanne.

6.2 Further co-development of Hardware and Software

Due to some delays in the distribution of the Oncilla hardware, testing of the above infrastructure on the hardware in order to create a fully vertical prototype, our experimentation on the hardware is yet limited. For first experimentation UniBi used the RoBoard RB 110, which is the embedded PC of the Oncilla robot, to run basic CCA setups and experiments. Yet without involvement of the motor controllers and hardware, we already tested the software architecture, conducted some benchmarking (cf. Section 5.1.3) and set up a compilation tool chain for optimizing the software architecture for the embedded PC. Experiments and benchmarking on the RoBoard influenced the choice of the technology mapping (fast inprocess transport, small footprint, lightweight) we did for the AMARSi Compliant Control Architecture.
To further improve co-development of, UniBi and EPFl came up with a coherent plan for the integration of software and hardware. UniBi will get a partial Oncilla setup to do extended testing of communication and library structures including motor control boards and mechanical hardware. This will allow for further testing especially regarding real-time constraints exposed by the motor control.

EPFL will work in the device driver level \textit{libSBCP}, which does the real-time motor control. UniBi will build the RCI representation \textit{RCIOncilla} together with EPFL. \textit{RCIOncilla} will use \textit{libSBCP} and has to carefully comply to the given real-time restrictions. It will provide an object-oriented interface for high-level access like setting, getting joint angles, parameterizing CPGs, reading torques, etc. using RCI abstractions (Task 7.1) and its domain types (DTOs). It will probably also provide additional security layers within a real-time context. Functionality of the \textit{RCIOncilla} layer will be exposed for local (in-process) clients and remotely (streaming) where clients can connect via the network. Ideally, the same interface is used such that components can be compiled for the robot or on a remote workstation without code change (cf. Figure 6.1).

Visits between of EPFL and UniBi partners to tighten co-development of software and hardware are already agreed upon and will appear during December 2011 and January 2012.

6.3 Example: Combination of Skills

To link the development of the AM ARSi Compliant Control Architecture with our AMARSi partners we evaluated the technical concepts and ideas of our component architecture as described in sections 4 and 5 in a vertical prototype (yet missing the hardware). The prototype was developed together with EPFL A who conducted a scenario which combines different learning skills in one architecture.[24] The scenario is the \textit{Oncilla} robot locomoting, then stopping and reaching for an object. For a scenario like this, we need a hierarchy which is able to control locomotion, reaching, and be able to smoothly switch between them with right timing. This is done using Dynamical Movement Primitives (DMP), combining locomotion control and reaching control, and called \textit{Unit Pattern Generator} (UPG). The Combination of Skills scenario is a vertical prototype in the sense that it touches all parts of the software architecture from higher-level sequencing to lower-level joint space control, yet missing the link to the real hardware. Note, that for a complete vertical prototype, the actual robot hardware needs to also be included to incorporate low-level (real-time) motor control into the vertical prototype. This will follow as next steps according to Section 6.2.

The reaching controller is based on a VITE (Vector Integration To Endpoint) model, generating a joint angle trajectory with a bell-shaped velocity profile for a given cartesian reaching goal. The oscillating movement for locomotion (trotting) is generated by a Hopf oscillator. The characteristics of the Hopf oscillator are that by setting the amplitude to a negative value, a Hopf bifurcation occurs and the attractor of DMP (which is a limit cycle) becomes an asymptotically stable fixed point attractor located at given goal. In the combination with the reaching controller, which provides the goal in joint space, a combined periodic-discrete controller is
created. So for a locomotion movement, amplitude of the Hopf oscillator is set positive. For the reaching movement the amplitude is set to (or slowly transferred to) a negative value and the reaching controller is executed to provide the according joint space task.\[24\]

Figure 6.3 shows the decomposition of this vertical prototype as we did for implementing it in the Compliant Control Architecture. The quadruped robot is represented by the resource nodes (see Section 4.4.1) QuadKneeNode, QuadHipNode and QuadTrunkNode. For this example we reduced robot control to joint position control. This means, the trunk node is not active during execution, QuadKneeNode and QuadHipNode (each existing four times) report current joint angles and awaiting joint angle commands.

The application is represented through five different CCA components that were implemented for this example, but already provide re-usable components:

1. **Sequencer**: The sequencer defines the (yet hard-coded) sequence of the application. The sequencer is timed (component configured with TimedProcessing strategy) and
6 Vertical Prototype

sends DTOs to the following components to trigger actions. This is sending a cartesian position to the reaching controller and sending an amplitude to the Hopf oscillator to blend between locomotion and reaching.

2. **Inverse Kinematics**: The inverse kinematics component calculated two joint angles for a given two-dimensional cartesian task space command (in this case sent by the sequencer). This component is re-usable and just requires a different configuration of the kinematic chain to be also usable for other limbs.

3. **Reaching Controller**: The reaching controller component wraps a one-dimensional joint angle controller. In this case we use a VITE (*Vector Integration To Endpoint*) model, generating a joint angle trajectory with a bell-shaped velocity profile for a given cartesian reaching goal. In this application we use two reaching controller components, one for the hip and one for the knee angle. This component is re-usable for any kind of joint position control.

4. **Hopf Oscillator**: The Hopf oscillator component wraps and oscillator, which oscillates while having a positive amplitude and bifurcates on a negative amplitude. Output of this component is the phase.

5. **DMP Transformation System**: The DMP component creates a joint angle movement based on a trained movement shape, the phase given by the Hopf oscillator and the joint angle targets given by the two reaching controllers. Output of this component is directly passed to the resource nodes as command.

**DMP Transformation System**

To be more concrete we show one of the involved Adaptive Components of this example as prototypical implementation as CCA Adaptive Components skeleton. This will show the implementations of the introduces concepts and the concepts conceptually formulated in [6]. The following section explains the parts of the Adaptive Component DMP Transformation System, which is a concrete instantiation and configuration of the Tracker skeleton, depicted in Section 4.4.4.

The parts connected by the Adaptive Component skeleton are:

1. **ODE** – The *ordinary differential equation* part of the adaptive component (cf. [24]).

2. **Adaptive module** – The Adaptive Module contains the ODE and in general initializes and/or adapts the ODE. In the present implementation of the example the component loads the ODE from a persisted file.

3. **Tracking Criterion** – The Tracking Criterion calculates the difference between the Adaptive Modules output and the actual joint angles feedback to indicate, if the tracker is converged.

Note: In this example no mapping or transformation components are necessary in the Tracker skeleton.
Figure 6.3: The DMP Transformation System Component modeled in CCA.

Ports of the components within this Adaptive Components Tracker skeleton are pre-configured by the skeleton and can be re-configured by the user to adapt it to its setup. A pre-configuration of ports may look like this:

1. **Phase Input**
   
   `/example/dmp/in/phase`
   
   Incoming phase from the canonical system. In the present example the output of the Hopf Oscillator. Port could be local or remote.

2. **Joint Angle Command Input**
   
   `/example/dmp/in/cmd/jointangles`
   
   Joint angles command around which the DMP system is oscillating. In the present example the output of the Reaching Controller.

3. **Joint Angle Feedback**
   
   `/example/dmp/in/fdb/jointangles`
   
   Joint angles feedback coming from the robot.

4. **Joint Angle Command Output**
   
   `/example/dmp/out/cmd/jointangles`
   
   Joint angles command to be sent to the resource nodes. So in the end the actual joint angles the motor controllers get as commands.

5. **ODE Status**
   
   `/example/dmp/status/ode`
   
   Status of the ODE (in its internal state space: converged, converging, . . . )

6. **Adaptive Module Status**
   
   `/example/dmp/status/am`
6 Vertical Prototype

Status of the Adaptive Module (is it learning (online, batch), is the cmd in learned area)

7. Adaptive Component Status
/example/dmp/status/ac
Status of the Adaptive Component (based on 5. and 6.): Is the tracker converged, is it not converged but in the training area, is it outside the training area).

The Adaptive Component Status port is probably a scope a sequencer would listen to. A sequencer or any other component could also listen to the superscope /example/dmp/status to get the entire, accumulated status or even to the superscope /example/dmp to get every data related to the component. This depends on the naming scheme of scopes in this skeleton and might change in future applications. The naming scheme of scopes to become more strict when using tools configure an application and chose scopes automatically.

The Combination of Skills prototype in CCA was proven to be running and used for experimentation at the Hands-On Oncilla Workshop in October 2011 in Lausanne.
7 Summary and Next Steps

Within this deliverable, we reported on the current state in work package 7 dedicated to the development of a metamodel and component architecture for AMARSi with a particular focus on a seamless integration of machine learning components for robot control. The deliverable was structured along our current working steps which aim to establish a model-driven development process eventually facilitating research at the level of motor skill architectures and not on single systems with a massive amount of hidden assumptions.

Section 3 presented an early draft of the architectural metamodel which fuses the insights on the functional architecture with software architecture concepts which shall eventually lead to a domain-specific language for the specification of full AMARSi system architectures. Subsequent to this, Section 4 introduced the current programming model which already reflects a larger set of the metamodel elements in its API design. This API and the chosen software concepts and implementation strategies which were discussed in Section 5 were already provided to and validated by the partners at an initial hands-on workshop. This early involvement of the different partners will continue, which is also reflected in our aim to continue work on a vertical prototype, cf. Section 6. While so far work on this prototype was based on simulation, we will gradually integrate the hardware with the software architecture which will be facilitated by mutual exchanges with the EPFL and IIT partners planned for the next months.
References


References


52
References