Best Practice in Robotics

Grant Agreement Number: 231940

Funding Period: 01.03.2009 – 28.02.2013

Instrument: Collaborative Project (IP)

Deliverable D-2.2: Specifications of Architectures, Modules, Modularity, and Interfaces for the BROCRE Software Platform and Robot Control Architecture Workbench

Author names:
Gerhard K. Kraetzschmar
Azamat Shakhimardanov
Jan Paulus
Nico Hochgeschwender
Michael Reckhaus
Bonn-Rhine-Sieg University (BRSU)

Lead contractor for this deliverable:

Due date of deliverable: 31.08.2010
Actual submission date: 31.08.2010
Dissemination level: PU/RE
Revision: 1.0
Abstract

Deliverable D2.2 provides specifications for the research and development work performed in work package WP2 of the BRICS project. The main purpose of the deliverable is to provide a document that allows experts inside and outside of the consortium to review the architectures, structures, and interfaces for the implementation work planned and ensure that constructive comments and criticism can be accounted for within the project duration.

The work described here is closely interdependent with the work in work packages WP3 (Algorithms) and WP4 (Tool Chain). In order to describe these dependencies and provide an understanding of how the various pieces fit together, this document first provides a comprehensive overview and survey on BRICS RAP, a reference process model for the robot application development process proposed by us. While numerous software frameworks, sometimes associated with specific development tools, have been developed for robotics, none of them comes with its specific development process model, or at least an adaptation of a well-known general software development process model. To our knowledge, BRICS RAP is one of very few, if not the only process model that accounts for a suitable balance of hardware and software development aspects, especially in robotics. Existing process models mostly focus on either one of these two complementary aspects.

We also describe a set of artefacts (mostly models of some kind) and a comprehensive tool chain; both are inferred from the process model and would jointly provide suitable support for the complete reference process. In practice, projects may need to exercise only parts of this process model, e.g. because the artefacts produced in a particular phase are already given, or because deployment and maintenance are decided to not be required for a research prototype. The reference process model is to be understood as a guideline for future research and development, as providing all the tools defined by the process is clearly out of scope of a single research project like BRICS.

We also propose a generic structure for the software architecture of a robot application, independent of the actual functional architecture. The architectures, modules, and interfaces for the BROCRE software platform and the Robot Control Architecture Workbench, as far as they can already be determined, then almost follow from this context. The report finishes with a summary of what could already be specified and a list of things that remain to be done in the upcoming project phase.
Acknowledgements

The research leading to these results has received funding from the European Community’s Seventh Framework Programme (FP7/2007-2013) under Grant Agreement No. 231940.
Contents

1 Introduction .................................................. 1
   1.1 The BRICS Project Context ...................................... 1
   1.2 Motivation for a Specifications Document .......................... 2
   1.3 Objectives of the Specifications Document .......................... 2
   1.4 Overview on the Report ........................................ 2

2 BRICS RAP .................................................. 3
   2.1 Phase 1: Scenario Building ...................................... 4
      2.1.1 Step 1-1: Scenario Definition .................................. 5
      2.1.2 Step 1-2: Scenario Generalization .............................. 5
      2.1.3 Step 1-3: Scenario Simulation Modeling .......................... 5
      2.1.4 Step 1-4: Customer Acceptance Test Definition ................. 5
      2.1.5 Notes on Phase 1 ........................................ 5
   2.2 Phase 2: Platform Building ..................................... 6
      2.2.1 Step 2-1: Hardware Platform Configuration ....................... 6
      2.2.2 Step 2-2: Software Platform Configuration ........................ 7
      2.2.3 Step 2-3: Robot Emulation Model Definition ....................... 7
      2.2.4 Step 2-4: System Component Testing ............................ 8
      2.2.5 Notes on Phase 2 ........................................ 8
   2.3 Phase 3: Capability Building .................................... 9
      2.3.1 Step 3-1: Component Construction .............................. 10
      2.3.2 Step 3-2: Skills and Capabilities Building ...................... 11
      2.3.3 Step 3-3: Content Creation ..................................... 11
      2.3.4 Step 3-4: Skills and Capabilities Testing ....................... 12
      2.3.5 Notes on Phase 3 ........................................ 12
   2.4 Phase 4: System Building ..................................... 12
      2.4.1 Step 4-1: Subsystems Assembly ................................. 13
      2.4.2 Step 4-2: Control Architecture Construction ..................... 14
      2.4.3 Step 4-3: Orchestration ....................................... 14
      2.4.4 Step 4-4: System-Level Testing ................................ 14
      2.4.5 Notes on Phase 4 ........................................ 14
   2.5 Phase 5: Benchmarking ....................................... 15
      2.5.1 Step 5-1: Components, Skills, and System Stress Testing ....... 15
      2.5.2 Step 5-2: Safety and Security Checks .......................... 15
      2.5.3 Step 5-3: Reliability and Durability Assessment ................. 15
      2.5.4 Step 5-4: Performance Testing ................................ 15
   2.6 Phase 6: Deployment ....................................... 16
      2.6.1 Step 6-1: Target Platform Component Identification .......... 16
      2.6.2 Step 6-2: Target Platform Resources Allocation ................ 17
6 Conclusions

6.1 Summary of Current State ............................................. 51
6.2 Future Work ............................................................ 52
List of Figures

2.1 The BRICS Robot Application Development Process. . . . . . . . . . . . . . . . . . 3
2.2 The scenario building phase of the BRICS robot application development process. 4
2.3 The platform building phase of the BRICS robot application development process. 6
2.4 The capability building phase of the BRICS robot application development process. 9
2.5 The system building phase of the BRICS robot application development process. 13
2.6 The benchmarking phase of the BRICS robot application development process. 15
2.7 The deployment phase of the BRICS robot application development process. . . . 16
2.8 The maintenance phase of the BRICS robot application development process. . . 17

3.1 First part of the tool chain for the BRICS RAP scenario building phase. . . . . . 20
3.2 Second part of the tool chain for the BRICS RAP scenario building phase. . . . . 20
3.3 The tool chain for the BRICS RAP platform building phase. . . . . . . . . . . . . 22
3.4 The tool chain for the BRICS RAP capability building phase. . . . . . . . . . . . 24
3.5 The tool chain for the BRICS RAP system building phase. . . . . . . . . . . . . . 25
3.6 The tool chain for the BRICS RAP benchmarking phase. . . . . . . . . . . . . . . 26
3.7 The tool chain for the BRICS RAP capability building phase. . . . . . . . . . . . 27

4.1 The architecture aspects used in BRICS. . . . . . . . . . . . . . . . . . . . . . . . 31
4.2 The different system abstraction layers used in BRICS. . . . . . . . . . . . . . . . 33
4.3 Organization of robot application software in a file hierarchy. . . . . . . . . . . . 39

5.1 Organization of BRICS software in a file hierarchy. . . . . . . . . . . . . . . . . . 44
5.2 The BROCRE software maturity model. . . . . . . . . . . . . . . . . . . . . . . . . . 48
Chapter 1

Introduction

The work described in this report was performed in the context of the EU project BRICS. We briefly describe this project context, then motivate the need for a specifications document and what the objectives of these specifications are. Finally, we overview the structure of this report.

1.1 The BRICS Project Context

BRICS\(^1\) addresses the urgent need of the robotics research community for common robotics research platforms, which support integration, evaluation, comparison and benchmarking of research results and the promotion of best practice in robotics. In a poll in the robotics research community performed in December 2007 95\% of the participants have called for such platforms. Common research platforms will be beneficial for the robotics community, both academic and industrial. The academic community can save a significant amount of resources, which typically would have to be invested in \textit{from scratch developments} and \textit{me-too approaches}.

Furthermore, scientific results will become more comparable which might promote a culture of sound experimentation and comparative evaluation. Jointly specified platforms will foster rapid technology transfer to industrial prototypes, which supports the development of new robotics systems and applications. This reduces the time to market and thereby gives the industrial community a competitive advantage. To achieve these objectives the BRICS project proposes the development of a design methodology, which focuses on three fundamental research and development issues. This methodology will be implemented in three interwoven lines of technical activities:

- Identification of best practice in robotics hardware and software components
- Development of a tool chain that supports rapid and flexible configuration of new robot platforms and the development of sophisticated robot applications
- Cross-sectional activities addressing robust autonomy, openness and flexibility, and standardisation and benchmarking

The authors of this report all work at Bonn-Rhine-Sieg University of Applied Sciences (BRSU), which is the partner in BRICS responsible for \textit{Architecture, Middleware, and Interfaces}. This work package is to provide fundamental software components using state-of-the-art software technologies and the usage of these components needs to be well embedded into the tool chain.

\(^1\)This section is a modest revision of the \textit{BRICS in a nutshell} section of the project proposal.
1.2 Motivation for a Specifications Document

Software development for robotics applications is a very time-consuming and error-prone process. In previous work [125] we identified hardware heterogeneity, distributed realtime computing, and software heterogeneity as major sources responsible for the complexity of software development in robotics. When, as is the case for BRICS, the objective is to speed up the development process, then measures should be taken to tame the difficulties arising from these problem sources.

One serious consequence of the above-mentioned problem sources is that many major innovative and complex service robot applications, e.g. in many EU-funded projects, seem to be built almost completely from scratch, with little or no design or code reuse from previous projects (see e.g. [73] [59] [60] [58] [62]). Software modules with almost the same functionality have often completely different interfaces when developed in different projects (see e.g. [26] [27] [50] [51] [129] [118] [28] [127] [126] [33] [123] [4] [48] [91] [97]). The architectures, even when designed for the same application, are often different to an extent, that even an expert developer needs substantial time to understand the architecture designed by someone else.

We conjecture that it is both necessary and possible to decouple functional and software technological aspects in a much more systematic manner. We acknowledge that the need to design very different functional architectures exists for the foreseeable time. We claim that a much higher degree of harmonization on the software level is possible (and necessary), and this can be the key to achieve much shorter development times.

1.3 Objectives of the Specifications Document

The main objective of this report is to clarify the relationships between i) the robot application development process, ii) the set of artifacts produced during the development process, iii) the tool chain to support the development process, iv) and the actual application software. This clarification should also help to better understand the different perspectives of the stakeholders involved, which include (at least) providers of hardware (sensors, manipulators, mobile bases, etc.) and software components (algorithms) for robotics, system integrators, application developers, and customers, all of whom may have or work in an industrial or academic background.

Based on this understanding, we want to derive the implications for the architectures of robot application systems, their modularization into (reusable) components, and the interfacing between these components. Where necessary, these notions must be clarified and structured first; this holds especially for the notions of “architecture” and “component”.

1.4 Overview on the Report

The remainder of the report is structured as follows:

- After this introduction, the chapter on BRICS RAP, provides a description of the proposed robot application development process.

- Chapter on BRIDE derives requirements for the tool chain that should be developed to optimally support BRICS RAP.

- Chapter on BRASA specifies a generic architecture for robot software applications.

- Chapter BROCRE presents specifications for the BRICS Robotics Open Code Repository, which is designed to play an essential role in improving software reuse in robotics.

The final chapter draws conclusions and describes implications for our future work within BRICS.
Chapter 2

BRICS RAP: The Robot Application Development Process

The Robot Application Development Process in BRICS (BRICS RAP or simply RAP) is a holistic process model for developing robotics applications in both academic and industrial settings. An initial model has been proposed by our group during the preparation phase of the BRICS project. The current proposal includes revisions accounting for feedback from project partners and experts of the field. It combines ideas from traditional software engineering [15] [114] [95] [90], agile software development [108] [81] [37] [110] [39], model-based engineering [72] [104] [113] [84] [18] [88], and industrial project management [[],[],[],[],]. [74] [20] [80] [133] [65]. A survey illustration of the process is given in Figure 2.1. BRICS RAP foresees seven different phases, each of which requires several steps to complete the task. The process phases and steps are briefly explained in the subsections following.

![Figure 2.1: The BRICS Robot Application Development Process.](image)

In order to fully understand the holistic nature of BRICS RAP, we assume that a customer, who is interested in acquiring a service robot application, either for an industrial or a residential
2.1 Phase 1: Scenario Building

When customer and system integrator meet at the beginning of a project, the most important issue is to make sure the system integrator gets a good understanding of the problem. This is in complete analogy to starting a (non-trivial) software project. As is the case in classical software engineering, the customer often does not really know what he wants. In robotics in particular, customers often have naive assumptions about what is possible with the current state of the art and what is not, and it is especially difficult to judge where precisely the limits are.

These difficulties are reflected by the steps in this phase (see Figure 2.2). After defining some initial scenarios, there are two steps to remedy the problem. Also, there is a step to explicitly define customer acceptance tests at this early stage of the project. Together, these steps constitute development steps elsewhere known as requirements engineering [115] [138] [16] [2] [92], use case modeling [36] [10] [100], or user story definition [38].

Figure 2.2: The scenario building phase of the BRICS robot application development process.
Chapter 2. BRICS RAP  

2.1. Phase 1: Scenario Building

2.1.1 Step 1-1: Scenario Definition

Some typical scenarios for the target application are described. The description preferably should include all information that is potentially relevant for the design and implementation of the target robot application, including but not exhaustively the environment, in which the application is supposed to operate, the relevant objects, the relevant subjects and their behavior, and the dynamics of the overall environment.

2.1.2 Step 1-2: Scenario Generalization

In order to avoid development of brittle applications [35] [85] which break down as soon as one of the tacit underlying assumptions is violated, this step focuses on the generalization of the scenarios. This means that the variability of scenario elements (environments, objects, subjects, behaviors, dynamics) must be discussed and described.

2.1.3 Step 1-3: Scenario Simulation Modeling

A simulation model for the scenarios is build here. Simulation is considered a helpful instrument for development [54]. The simulation modeling step can be done simultaneously with steps 1 and 2, especially if an interactive simulation model building tool is available, which allows the immediate visualization of the environment model under development. Such a visualization tool can thereby be of great help in the scenario building and generalization process.

2.1.4 Step 1-4: Customer Acceptance Test Definition

The range of possible combinations of scenarios and expected services by the target robot application can quickly grow to dimensions prohibitive for exhaustive testing. The definition of customer acceptance tests [19] [83] allows the customer to express relative importance of different end-user functionalities. This information gives developers directions for devoting attention to particular system functionalities. Aside of acceptance tests made available to developers, the customer should define for each test one or more variations, which will not be made available to developers, in order to test the robustness and variability of the developed solutions.

2.1.5 Notes on Phase 1

Agile Development Aspects: The steps in this phase are expected to be executed repeatedly. Every task, which the final robot application is expected to perform, should be reflected in at least one scenario (user story). During this repetitive process, the steps in this phase do not necessarily need to be executed in each iteration, or in the order presented here. For example, if a scenario involves grasping an object, and the range of the objects that the robot should be able to grasp has already been elicited in the scenario generalization step in a previous iteration, it is not necessary to repeat it again. However, requirements are acquired from people, and people are not always good in thinking of and listing all requirements right away. Looking at some problem aspect again, possibly from a different problem angle, often reveals extra requirements and helps to gain a more complete picture. Therefore, making it an explicit exercise to consider generalization for each new scenario may help to obtain good coverage of the overall problem at an early stage.

Model-Driven Development Aspects: Three possible models can be identified:

1. a model for a scenario description,
2. a model for a simulation model, and
3. a model for a customer acceptance test.

The scenario description model should foresee a structure of a scenario description, including for optional elements, and have explicit indicators that need to be looked at during the scenario generalization step.

2.2 Phase 2: Platform Building

The platform building phase foresees steps necessary to prepare the actual software development of the target application (see Figure 2.3). The hardware platform needs to be configured, from which the software platform can be derived, which provides the necessary device drivers and various utilities. Furthermore, a robot emulation model is derived to complement the simulation model defined during scenario building. The purpose of this step is to allow software development independently of the actual availability of the hardware platform, which due to manufacturing scheduling constraints often may take several months to build. As in all steps, testing procedures are foreseen as well.

![Figure 2.3: The platform building phase of the BRICS robot application development process.](image)

2.2.1 Step 2-1: Hardware Platform Configuration

Once a good understanding exists of which tasks the target robot application needs to solve and how the environment and task execution context looks like, a hardware platform can be and should be configured. Currently, it seems unlikely that much sensible software development can take place without knowing what the target platform will be.

For the configuration process itself, several approaches are possible. The most viable within the BRICS project seems to assume a small set of generic robot hardware platforms [117], for which a limited number of components can be configured. For an industrial mobile manipulator from KUKA, for example, one might choose between two base variants consisting of the KUKA OmniRob mobile platform [76] with either one or two KUKU LWR lightweight arms [64] [75] mounted on it. Other configuration choices may pertain to the computational hardware to be integrated, and the number of sensors (laser range finders, cameras, etc.) to be mounted and their placement on the robot.

At least the vast majority of hardware requirements should be inferable from the information acquired in the scenario building phase. It should be no problem, however, to modify the hardware configuration later on, e.g. if the need for an additional sensor arises or an initially selected hardware component needs to be replaced by another one.
2.2.2 Step 2-2: Software Platform Configuration

The software platform is the complete software foundation on which the actual robot application software will be built. The software platform consists of

- the operating systems and/or any required firmware for the computational devices configured into the hardware (embedded PCs, laptops, microcontroller boards, etc.),
- the device drivers for all relevant hardware devices, including those for all sensor and actuator devices,
- configuration information that can be derived from the hardware configuration, e.g. coordinate translations from a camera or laser range finder coordinate system to the coordinate system of the mobile base,
- utilities and tools for configuring, calibrating, operating, monitoring, logging, and analysing all hardware components,
- software technologies components\(^1\) required for operating the robot application software, such as communication middleware, user interface toolkits, or libraries supporting the integration and use of particular hardware devices,
- all the tools and facilities that come with or are necessary for an integrated development environment, including editors, compilers, debuggers, profilers, etc., and providing support for both model-driven development and agile development,
- libraries of best practice algorithms that could potentially be reused in building the application, and
- libraries of reusable software components of all of the above, usually the result of a componentification process.

The suite of software packages that will make up the software platform is determined by information from three different sources:

1. Information directly inferred from the hardware configuration. Example: device drivers for sensor components.
2. Information representing customer choices. Example: operating systems to be run on computational hardware, or a particular environment for building the user interface
3. Information following from application developer decisions. Example: a middleware package selected for distributed programming support.

2.2.3 Step 2-3: Robot Emulation Model Definition

Simulation plays an important role in BRICS RAP. It not only allows for software development without actual access to the robot hardware (e.g. because the hardware requires time to build, or access to the hardware is restricted or constrained due to resource limitations), but also provides a means to safely test a design in simulated extreme situations or to assess new designs with unknown safety properties. The simulation model built in the scenario building phase needs to be complemented by a robot emulation model to allow for such simulation runs. We prefer the use of robot emulation model vs robot simulation model to emphasize that the simulation should allow to test the targeted robot application software without changes, i.e. only the interfaces to hardware components should be replaced by emulated versions of these components [30] [68] [17].

\(^1\)Component is here used in an informal sense, denoting a piece of software that becomes part of robot application or the environment required to execute it.
2.2.4 Step 2-4: System Component Testing

At the end of the platform building phase it should be possible to perform tests on all hardware devices, software components, and development tools that were selected [111] [47] [23].

2.2.5 Notes on Phase 2

Agile Development Aspects: The platform building phase can be executed in a very agile manner, especially if all steps are supported by appropriate tools, that allow fast iterations of the process and performing only small pieces and steps in each iteration. After selecting a mobile base and some default computational device on it in Step 2-1, an initial software platform and an emulation model can be configured and generated in Steps 2-2 and 2-3. Some mobility tests can be defined, and executed using the emulated robot in the simulated environment, and a manual remote control tool. The next iteration may extend the hardware configuration by adding some sensors, e.g. two laser range finders. Different options for mounting them could be explored by cycling through Steps 2-1 to 2-4 repeatedly. Each iteration is performed with a clear goal in mind (exploring a design option) and yields clear results that are both useful and required by successive steps. The idea of decomposing the development in many small, decoupled steps while always keeping things integrated is one followed closely by the agile development community. The strict test orientation ensures that previously achieved goals and features remain functional despite the many small changes made to the system.

Model-Driven Development Aspects: The platform building phase provides an excellent example of the whole idea of model-driven development: both the software platform and the robot emulation model are generated from a hardware model which is specified in Step 2-1. For the platform building phase, the hardware model defined in Step 2-1 represents the platform model of an MDD process, the software platform generated in Step 2-2 represents the platform-specific model of an MDD process. The platform-independent model of an MDD process is represented by a generic BRICS robot application software architecture (BRASA), which is subject of a later chapter.

After the two initial phases a point is now reached where the actual core software development activity can start. The robot hardware is determined. Even if it is not available yet, we can work with its emulator. All the hardware-related software components are there and tested, as well as a lot of utilities and tools needed for software development.

In retrospective, it seems that many development projects in robotics seemed to assume that they start at this point, and that everything their project required up to that point is merely a matter of acquiring a robot platform and installing some development environment. Projects that plan to develop new robot hardware usually tend to underestimate not only the effort for hardware development itself, but especially the effort for providing all the tools and utilities to provide an adequate development environment to software developers. If such projects are funded only for a limited period of time, it is almost a necessity that they eventually fall short of achieving their initial goals, because the delays caused by either the late arrival or continued unavailability of the development platform (as defined here) can never be compensated for later on. Even projects which choose to select an almost complete mobile platform from a vendor and “only” add a few extra sensors or a mobile arm often suffer from this problem, as the software implications of these “small additions” to the hardware platform, lying at the borderline between the hardware platform-oriented developers and software functionality-oriented developers, are easy to underestimate. Even projects who initially opt for a complete vendor-based solution consisting of a mobile platform and associated development environments, may run into the problem...
that the developers find the software environment inadequate and too limited for their purposes, and end up in investing lots of effort into developing a “suitable” development environment.

In all of the above scenarios, a lot of re-invention of the wheel is happening; too much in the opinion of many experts. BRICS RAP also targets to remedy this problem, mainly by accepting the fact that an application development process can be made much more efficient, if the right environment and tools are provided to the developers, including support for re-use of software in the form of readily accessible libraries.

2.3 Phase 3: Capability Building

After completing phases 1 and 2, the stage is set for the actual core software development, which will happen in phases 3 and 4. The BRICS project has decided to adopt the idea of component-based software development [109] [98] [101] [52] [11] [61]. It seems to best foster software reuse [78], and supports both ideas from model-driven development [72] [104] [113] [84] [18] [88] and agile software development [108] [81] [37] [110] [39].²

![Figure 2.4: The capability building phase of the BRICS robot application development process.](image)

The focus of Phase 3 is on building functional capabilities (see Figure 2.4). As essential major steps, we foresee activities aimed at making hardware devices and algorithms available as re-usable components, as well as composing components to create particular capabilities such as recognizing people, perceiving objects, understanding speech, producing speech, exploring and mapping the environment, planning paths, executing trajectories, grasping objects, making gestures, etc. The focus of Phase 4 is on composing complete systems/applications from such component capabilities and ensuring that they work together correctly and interact in the desired ways. As components play a central role in both phases, there seems to be significant overlap between them. Although this may be true in terms of the technologies and tool sets used, the difference can be characterized as follows:

- The capability building phase concentrates on developing isolated functionality. It takes a component perspective and provides a piece of software that is designed for integration into larger systems. If the capability component itself is composed of components, these components usually interact in relatively simple ways. Usually, there are no or few resource conflicts on the component level. Any kind of user interaction is considered in an isolated way without assuming a particular context.

- The system building phase focuses on developing integrated, system-level functionality. It takes a system perspective and provides a complete system or application, which can

²An overview of the software architecture for the target robot application is given in Figure 4.2.
run standalone and does not depend upon integration into a larger system. The system building phase will usually have to integrate a multitude of capabilities, and this integration may produce many and serious resource conflicts. Resolving these may imply non-trivial modifications to the components or the introduction of specific components for conflict resolution. Particular care must be taken to integrate and consolidate user interaction such that the overall system behavior is understandable and predictable for the human users and operators.

2.3.1 Step 3-1: Component Construction

Component-based software development works with components. As will become clear later in this document, we view component-based software development as a means to structure and manage large, distributed applications, i.e. predominantly for programming-in-the-large, and object-oriented programming as a means for programming-in-the-small, which should be applied for the implementation of components itself. Non-object-oriented programming is acceptable only for legacy code and should be avoided as much as possible.

This view basically defines the activities to be done in Step 3-1: to turn whatever initial ingredient we get (a device driver, an implementation of an algorithm) into a component. For a hardware device, these activities include

1. installing, configuring, and running whatever vendor-supplied software comes with the device, especially the respective device drivers, any libraries that may be supplied for processing sensor data produced by the device, or any controller that may be supplied for controlling an actuating device,
2. selecting or defining interfaces for the device and data structures used by these interfaces,
3. wrapping the vendor supplied code into usually one object-oriented class (using the interfaces and data structure previously defined),
4. providing the interfaces as network-transparent services by combining the object-oriented class with middleware functionality, and
5. encapsulating the network-transparent services into a re-usable component.

For an algorithmic component, usually the implementation of a particular functionality, like an object recognition algorithm, these activities include

1. analyzing and understanding the legacy code,
2. refactoring the legacy code or re-implementing the algorithm in an object-oriented fashion, as a set of classes,
3. harmonizing class interfaces and the data structures used by them, and assimilation to abstraction hierarchy,
4. providing the interfaces as network-transparent services by combining the object-oriented classes with middleware functionality, and
5. encapsulating the network-transparent services into re-usable components.

Developers may, of course, also develop components completely from scratch. In this case, there is no legacy code. Developers can directly implement the algorithm using object-oriented classes, interfaces, and data structures, turn these into network-transparent services, and turn these into components.
Remark: The computational aspects incorporated by a component are manifold, the required knowledge and background required for it is deep. There are few programmers out there who master all of low-level systems programming or embedded system programming, design and efficient implementation of algorithms from specific domains like computer vision, non-linear control, and task planning, in a state-of-the-art object-oriented programming language, appropriate use of communication middleware, and the implications of programming distributed systems in an equally competent and adequate manner. The componentification process is intentionally structured into several steps, each of which requires different skills and background knowledge, in order to make the process more manageable. We get more steps, but smaller cognitive load in each step.

2.3.2 Step 3-2: Skills and Capabilities Building

Once the elements needed to build an application are available as components, they can be combined and composed into more complex components which are able to provide a particular skill or capability for the robot. It is normally the first step where composite components are needed [132] [57] [102] [71] [42] [131] [139] [14] [5] [9] [44] [70]. We take a recursive view of composite components: Any composite component can be used as a component for building a more complex composite component. As we have outlined before, none of these components is supposed to be executable as a standalone application and may demand a certain context to be runnable.

2.3.3 Step 3-3: Content Creation

The content creation step may appear to be unusual. However, many robot applications involve functionalities the constructions or implementations of which often rely on databases (of non-trivial size), which are time-consuming to provide and themselves require some care in their production. Examples of such databases include:

- database of maps of large environments
- knowledge bases for objects to be recognized/manipulated
- knowledge bases for world models
- knowledge base of action and operator models for planners
- knowledge base of methods for hierarchical task network (HTN) planners
- database of speech samples for speaker recognition
- database of speech samples for speech generation
- database of faces for people recognition
- generally, databases of training examples for learning problems

While the type and format of the above data is often easily determined and implied by the choices of algorithms, the mere process of filling the knowledge and data bases can be very time-consuming. The quality of the data must be ensured, and inconsistencies, incompleteness, and incompatibilities often present severe obstacle for project progress.

For some of the above examples, current research indicates that such information could possibly be downloaded and updated from the Internet. The EU project RoboEARTH [120] [8] is an example for such research.
2.3.4 Step 3-4: Skills and Capabilities Testing

This step should be quite straightforward: for all intermediate steps and final outcomes of the activities in this phase tests need to be defined and executed. These tests will be particularly helpful in situations, where the complete system exhibits overall some strange behavior. With all the tests so far, developers can check system functionality bottom up from simple hardware devices to simple or advanced skills and capabilities [111].

2.3.5 Notes on Phase 3

Agile Development Aspects: Phase 3 allows for agile development in many ways. The independent consideration of numerous individual skills and capabilities allows for concurrent development of small chunks of software in shorter periods of time. All steps in Phase 3 define tasks that can be delineated from each other quite easily. Bottom-up, top-down, inside-out, and incremental refinement are all approaches that easily fit with the activities described. Test-driven development is also possible; in this case one would start with Step 3-4, then iterate through the steps of the phase.

Model-Driven Development Aspects: Model-driven development aspects play an important role during Phase 3, as they bear substantial potential for accelerating robot application development. Although the detailed definition and concepts of models in this context are not yet completely settled, there is an agreement that software re-use should be the guideline for further development. Techniques that foster software re-use are the introduction of a component types and abstraction hierarchies not only on the object-oriented class level, but also for interfaces, data types, and components. This generates a drive for abstracting and harmonizing interfaces and standardizing data structures used in these interfaces. A high degree of platform independence, one of the essential goals in model-driven development, can be achieved by exploiting these abstraction hierarchies when defining skills and capabilities as composite components.

2.4 Phase 4: System Building

The fourth phase of the development process is concerned with both the functional and software integration of several skills and capabilities into more complex functionality and the building of complete systems. It is probably the least consolidated phase in terms of methods that are well established and agreed upon in the community. This problem was previously coined into the term “1001 architectures for 1000 robots” [94], in order to express the fact, that there are no commonly agreed upon approaches for designing a good functional robot control architecture and its implementation in a software architecture.

We assume that a general functional robot control architecture that is suitable for a wide range of applications does not exist. Such a functional architecture can possible be designed for a particular domain, or a particular type of robot application, but may require at least modifications and tuning for each application. Any software engineering process for robotics can only be successful if it allows developers to build customized functional robot control architectures. However, the software development process can be significantly improved, if there are established means to implement these functional robot control architectures in software architectures. Component-based development has the potential to provide such means [132] [57] [102] [71] [42] [131] [139]. A complete robot application can be viewed as a (potentially very complex) composite component. Other than an arbitrary composite component, it must be executable as a standalone application. Additional constraints may be defined that distinguish a full-fledged system from other composite components.
This phase is therefore primarily concerned with the development of (non-trivial) composite components (see Figure 2.5). The steps in this phase focus on particular aspects that arise in the composition process, in particular

- structural composition of a composite component from simpler components (in the following called elements),
- defining component-level control, and
- orchestration of component-level with element-level activities, including prioritization and conflict resolution.

These steps are closely related with each other; it may not always be possible or sensible in practice to clearly delineate from each other, but as at least the first three steps in this phase are expected to be performed by the same developers, this is currently not considered a problem. At the very least, the steps already identified clearly point to particular activities that need to be performed during the composition process. It is currently difficult to assess, to which extent these steps can be supported and/or automated by appropriate tools, and how much effort developers need to devote to them. We can foresee situations, where a particular way of structural composition can imply almost automatic generation of control structures with little or no need for orchestration. On the other hand, one could imagine compositions that raise difficult control problems and conflicts which may be very hard to resolve. Practice will show over time, what is really needed.

2.4.1 Step 4-1: Subsystems Assembly

The subsystem assembly step yields an advanced skill or capability by defining a composite component from several simpler components (called elements or subcomponents). The elements may themselves be composite components. The subsystem assembly step is worried only about structural issues. It defines the services provided by the component, the ports over which these services are made available, the interfaces governing the ports, and the data structures required for the interfaces; this defines the outside view of the new component. The subsystems assembly step furthermore identifies the subcomponents that make up the new component, and how they are connected, both between each other and with the component’s ports. Conflicts with respect to resources may occur, e.g. that two elements are foreseen, each of which needs a particular device as a subcomponent. Such structural conflicts have to be resolved, either by putting constraints on the connections between ports, or by introducing additional elements (e.g. one that broadcasts
data provided by a device to several consumers, or which applies some transformation on the data before delivering it, or which applies some filter, or some kind of arbiter).

### 2.4.2 Step 4-2: Control Architecture Construction

In order to understand our concerns with respect to control architectures, it is helpful to consider every component as potentially being executed in its own process or thread. Even if it is later decided to apply a different mapping of the software architecture to a runtime architecture, it is extremely helpful to take this view, as it forces developers to be more explicit about control issues, rather than hiding this implicitly in deeply nested procedure call hierarchies.

Saying something about the control architecture of a component means first of all specifying how the services provided over the component’s ports are constrained, e.g. how they can be sequenced, etc. This is often done by specifying a state machine. Furthermore the relationships with the control architecture of the elements must be defined. Basically, this step complements the previous one, which focused on structural aspects, by focusing on control aspects.

### 2.4.3 Step 4-3: Orchestration

Orchestration can be viewed as an effort to make the control architecture robust. Construction of the control architecture will usually focus on control aspects closely related to the services provided by the newly defined component. In this step, developers should look at potential control conflicts and resolve them, ensure correct synchronization between the component and its elements, and ensure the avoidance of problems typically arising in distributed applications, live locks, dead locks, starvation, trashing

### 2.4.4 Step 4-4: System-Level Testing

This step tests complex subsystems or the overall robot application [87] [89]. The customer acceptance tests should be successfully run.

### 2.4.5 Notes on Phase 4

**Agile Development Aspects:** It is not yet clear how agile development can be applied to this phase.

**Model-Driven Development Aspects:** While the above description basically takes a bottom-up view and seems to involve very complex design activities, we are positive that it will be possible to develop blueprint architectures, i.e. composite components defined in terms of abstract component elements and already predefined control architecture and orchestration mechanisms. In this case, the developers would have to match generic elements with concrete components in Step 4-1, but would not have to consider control and orchestration aspects.
2.5 Phase 5: Benchmarking

A robot is a potentially dangerous product, and special care must be taken to ensure it is complete, safe, reliable, and performant [79] [31] [19] [69]. Each of the benchmarking steps focuses on a different aspect.

![Phase 5: Benchmarking](image)

Figure 2.6: The benchmarking phase of the BRICS robot application development process.

2.5.1 Step 5-1: Components, Skills, and System Stress Testing

This step focuses on functionality. Tests should ensure that the components provide the requested functionality, including producing the correct results under typical and extreme operating conditions (e.g. variations of system load, co-occurrence of events, difficult environment conditions like heat, sun, darkness, water, etc.).

2.5.2 Step 5-2: Safety and Security Checks

The focus of this step is on safety and security. Safety is concerned with ensuring that the robot does not harm to subjects or objects. Security is concerned with ensuring that the robot cannot be accessed and abused by malevolent users or other systems.

2.5.3 Step 5-3: Reliability and Durability Assessment

Any product must be ensured to be sufficiently reliable and durable before it can be marketed.\(^3\) Once the robot application has reached a state where it provides the necessary functionality, developers must also assure its reliability and durability by running long experiments, under potentially extreme conditions.\(^4\)

2.5.4 Step 5-4: Performance Testing

Last but not least, this step checks maximum operation rates for each component, skill, subsystem. This step tests what is classically associated with (performance) benchmarking.

\(^3\)For example, some researchers who use Hokuyo laser scanners very intensively in somewhat rough operating conditions have recently reported serious wearout issues rendering the sensor defect after just a few months.

\(^4\)Willow Garage continuously runs tests of their prototypes over several days in a non-air-conditioned container exposed to the California sun. Temperatures inside the container are usually well over 40 degrees Celsius, often more.
2.6 Phase 6: Deployment

So far, we have developed the application, and tested it on a development prototype in the lab. It is now time to deploy the robot software application onto the actual target system, and to make sure we can operate and maintain the application later on when it has been delivered to the customer [22] [53]. The steps required for this process are illustrated in Figure 2.7.

![Figure 2.7: The deployment phase of the BRICS robot application development process.](image)

2.6.1 Step 6-1: Target Platform Component Identification

The focus of this phase is on the differences between the system that has been used for development so far and the system to be deployed to the customer. These differences may include differences in hardware and software.

If we assume that developers did a good job on configuring the hardware platform in Phase 2, and that we adapted this configuration during development as additional requirements appeared, how can then hardware differences occur between the development platform and the target deployment platform? We can identify mainly two reasons for such a situation to arise:

1. Development platforms are usually incomplete. A typical example for robotics is that many development platforms are built without a cover or hull, because it has no role in the development process. However, when a hull is finally added to the “finished” robot, it could suddenly behave very differently. The hull may constrain the actuators or affect the sensors in unexpected ways. Thus, a robot application should not be considered finished unless it has been fully assembled, including all (seemingly) non-functional parts, and has been tested in its final configuration.

2. Development systems often have a tendency to be somewhat lavishly with respect to resources. A good example are computational resources, i.e. the number of computers, and their sizing, configured into the hardware platform. While this may be okay during development, and may even speed up development, it usually creates a cost problem for the final product. Therefore, the hardware configuration needs to be critically reviewed and optimized during this step.

The software side of the target platform component identification also has to cover two aspects:

- Modifications to the hardware may induce changes to the software.
- Software components that were only needed during development, e.g. for debugging and profiling purposes, but are not needed during operation, can be removed.
Last but not least it should be mentioned that executable code for the target application will usually be optimized and compiled for speed, while development versions of the code will usually be compiled such that it can be easily debugged.

### 2.6.2 Step 6-2: Target Platform Resources Allocation

Once both the final hardware and software components have been determined (and potentially differ from the setup available during development), the allocation of software components on the target hardware platform needs to be determined and optimized.

### 2.6.3 Step 6-3: Maintenance Instrumentation

Another important step before delivery of robot application to the customer is its instrumentation for maintenance. This includes taking provisions for logging data that are required or at least helpful for maintenance, and adding and configuring tools for analyzing these data, tools for on-site (re-)calibration of system components, and tools for recording the maintenance operations themselves. Needless to say, access to all documentation, models, and source code for the whole application needs to be accessible during maintenance; the deployed system may itself have this information on local storage, or it must be accessible from a remote code repository.

### 2.6.4 Step 6-4: Target Platform System Testing

As the steps in the deployment phase may incur various modifications to the system, it must undergo another testing phase. This testing phase may include re-running tests defined in various previous phases, but especially if hardware modifications have been performed, the definition of additional tests for particularly the final hardware platform may be warranted.

### 2.7 Phase 7: Maintenance

Little is so far known about this phase, as there are so few successful robot applications out there. The steps suggested here (see Figure 2.8) are inferred from experience in other industrial projects, such as maintenance of a production line, or comparably complex products, such as an upper-class automobile.

![Figure 2.8: The maintenance phase of the BRICS robot application development process.](image)
2.7. Phase 7: Maintenance

2.7.1 Step 7-1: Log Analysis

Complex systems and complex products, which are controlled by computing devices, nowadays keep records during operations. These records include data about intensity and duration of system use, documentation of various events, like system mode changes, errors that have occurred, etc. These records usually indicate problems during system operation and allow to often allow to infer necessary maintenance activities, such as replacements of parts, the need for re-calibration, etc. The first step during maintenance usually consist of a readout of the recorded data and their analysis.

2.7.2 Step 7-2: System Tuning

System tuning is an activity that is typically performed after deployment of a system. Only after the system was operational for some time and information about its actual use has been acquired, it may be possible to apply modifications and improvements that help to avoid failure situations and to optimize the overall system performance.

2.7.3 Step 7-3: System Extension

Occasionally, small system extensions need to be performed during the maintenance phase. These extensions may be partially induced by replacement of parts (hardware devices), additional parts (additional sensors mounted to the robot), and extension of system functionality due to new requirements that have arisen during operation.

2.7.4 Step 7-4: On-Site Testing

Last but not least, the modified system needs to be tested before normal operation can continue. Due to a variety of constraints (time, space, and equipment available, environment usable, safety and security), the full range of tests that can be performed during development, especially the benchmarking phase, will normally not be executable during maintenance. The range of runnable tests (determined during the deployment phase) should be rich enough to ensure at least safety, security, and reliability. The tests actually executed during this step may be determined by the type and extent of maintenance work performed in the previous steps. For example, if log analysis indicated that the system seemed to have been working well and no serious failures were recorded, and no system tuning and system extensions have been performed, minimal or even none on-site testing at all may be required. If the maintenance work required modifications involving mounting additional sensors and/or mounting existing sensors at different places on the mobile platform, and significant modifications of the software were necessary, then a suitable range of tests should be run to ensure safe and reliable operation of the system.
Chapter 3

BRIDE: Tool Support for the Robot Application Development Process

The BRICS Robotics Integrated Development Environment (BRIDE) is an integrated development environment for the robotics domain. It is designed to especially support the BRICS robot application development process, and provides a set of tools to support the execution of the various BRICS RAP phases and steps. As the tool chain is the core subject of another work package, only requirements that arise from BRICS RAP and software architecture requirements are formulated here and a full functional specification of all BRIDE aspects is left to the partners involved in WP4. The discussions in that work package are currently focused on activities that fall into BRICS RAP phases 3 and 4. However, requirements arising from other process phases may influence the software architecture and the repository structure, and therefore it is essential that the tooling of the whole process is considered, even if these tools may be developed only much later.

Three important strategic decisions taken early in BRICS influence our considerations to a large extent:

1. adopting a model-driven software development approach [104] [113] [18] [88],
2. applying component-based programming in software development [109] [98] [11] [61].
3. using Eclipse as development environment [66] [1] [29] [99] [32]

The former two were already mentioned and explained where appropriate, and further details will be provided along the way. The latter means that all tools defined here can in principle be designed and implemented for the Eclipse development platform, mostly as Eclipse plugins, and reveal themselves to their users as special perspectives and views in Eclipse. As previously stated, we would like to emphasize here again, that the design and implementation of the complete tool chain is beyond the scope of the BRICS project, and the focus of BRICS, especially WPs 2, 3, and 4, will be on providing tools that strengthen the reusability of software components for robotics. The decision for Eclipse recognizes the effort that has been and continues to be devoted to Eclipse development and the level of dissemination and acceptance it has already achieved. Naturally, all tools could also be implemented as standalone tools or be integrated in some other major software development platform.

Each of the following sections will consider a BRICS RAP phase. Each section is equally structured into three subsections:

1. The first subsection gives an overview on how the inputs, outputs, tools, and users of these tools are connected and depend on each other, usually simply in the form of a graphical illustration.
3.1.1 Scenario Building Tool Chain

Figure 3.1: First part of the tool chain for the BRICS RAP scenario building phase.

The activities in this phase are illustrated in two separate parts. Figure 3.1 shows the scenario modeling and scenario generalization steps, while Figure 3.2 shows the activities for building simulation models of the scenario and for defining customer acceptance tests.

Figure 3.2: Second part of the tool chain for the BRICS RAP scenario building phase.
3.1.2 Scenario Building Artifacts

The tangible outputs produced in this phase should be:

- **Scenario models** (scenario description files), as outcome of the scenario definition step.

- **Generalization rules**, which can be used to generate *scenario model variations* of the customer-defined scenarios.

- **Simulation models**, usable by a suitable simulator, to run simulations of the scenario and to test the target robot application in customer-defined simulated environments. The simulation models can be partially generated from the scenario descriptions, but may need additional information provided by users/customers.

- Customer acceptance test definitions, to be used later on for testing the completed system. The acceptance test will need descriptions of scenes, e.g. for initial, intermediate, and goal achievement situations, and sequences of situation descriptions or plots to describe changes to situations. Suitable visualizations of these situations would be of great help and simplify communication with the customer.

No tangible inputs are assumed for this phase.

3.1.3 Scenario Building Tools

The tools support for the scenario building phase include:

- The scenario *modeler* should support a knowledgeable person, e.g. a sales engineer or application developer, who is interacting with a customer, to define the application scenario. The outcome, scenario models, should include information about the environment the robot is supposed to work in, the objects in this environment, all objects that need to be recognized, located, or manipulated, the occurrence of humans or other robots in the environment and their activities, etc. This tool should make use of a **scenario library**, which contains schematic descriptions of environments (helping the application engineer to make the scenario description as complete as possible), plus typical examples, e.g. from previous projects, that could be selected and simply modified to quickly get tangible results.

- Based on scenario models, and supported the scenario library, the scenario *generalizer* should help the application engineer and the customer to generate a set of generalization rules for scenarios. These rules include e.g. information about how much task-relevant objects can vary, allowable lighting variations in the environment, etc. The generalization rules, the scenario library, and the scenario models will be used by a **generalization engine** to generate variations of the scenario originally acquired with the customer.

- Using the simulation *model builder*, the scenario models (including their variations) must be turned into simulation models. Scenario models contain information relevant for the customer to describe the desired application, but they do not contain sufficient information to build simulations. A simulation library providing a wide range of simulation components for frequently occurring elements in scenario models can facilitate this process significantly.

- Defining customer acceptance tests at this stage of the project involves specifying detailed scenarios with particular initial, final, and potentially intermediate situation descriptions. Making use of the simulation model of the environment and the simulation engine, it is possible to create simulation logs from which scene descriptions, plots, and visualizations could be extracted.
3.2 Tools for Platform Building

3.2.1 Platform Building Tool Chain

The activities in the platform building phase are illustrated in Figure 3.3:

![Diagram of platform building tool chain]

Figure 3.3: The tool chain for the BRICS RAP platform building phase.

3.2.2 Platform Building Tangible Artifacts

The tangible outputs produced in this phase should be:

- A **hardware platform model** which describes all hardware components that are needed to assemble the robot application and how they are put together.

- A **software platform model** which initially contains information on the software counterparts of all hardware components, i.e., usually some kind of device driver to interface a component, plus all software utilities that may come with such hardware components (debugging tools, logging and monitoring facilities, etc.). Furthermore, information on the operating systems and other system-level software of the computational devices configured into the robot.

- A **robot emulation model** which complements the simulation model built in the previous phase to allow for emulation of the robot in simulated scenarios. The availability of suitable simulation environment is considered a key element of decoupling hardware and software development and thereby shortening overall development cycles.

- A set of **component tests** which allow to test-drive all elementary hardware components in the system configuration.

3.2.3 Platform Building Tools

The tools support for the platform building phase include:

- The hardware platform configurator allows to interactively configure complex robot systems. It needs a library of robot hardware descriptions. The configuration process is...
guided by the scenario models and customer acceptance tests. Every part configured into
the robot should be justified either by a particular scenario, or be a prerequisite of such
parts.

- The **software platform configurator** allows to interactively configure the basic soft-
  ware platform\(^1\) for the robot application. It makes use of a large repository of software
  components. While numerous components are automatically determined by the hardware
description model, there may be a choice between several software options (e.g. three
different implementations of interfaces). Other software choices to be made include the op-
erating systems (if any) for the computational devices, and the range of available software
utilities and tools that may be available for the selected hardware components.

- The **software platform generation engine** actually generates and configures the soft-
  ware platform based on the software platform model and using the robot software repos-
  itory. The software platform will include automatically configured tests for the hardware
  components.

- The **robot emulation generation engine** produces an emulation model of the robot,
  using the software and hardware platform models and the robot software repository.

- The more or less automatic generation of both the software platform and the emulation
  model is possible only if the components configured into the target platform have been
  previously described as a hardware component, and the respective software counterparts
  have been described and stored in the robot software repository. If a new component is
  to be added to the set of available choices during hardware configuration, then a software
developer needs to describe them accordingly using a **component builder**

- The **test builder** tool is (here) used to define suitable component tests for new hardware
  components.

### 3.3 Tools for Capability Building

#### 3.3.1 Capability Building Tool Chain

The activities in this phase are illustrated in Figure 3.4:

#### 3.3.2 Capability Building Artifacts

The tangible outputs produced in this phase should be:

- A set of **software components** implementing particular skills and capabilities, including
  models for these components.

- A set of **component tests** that allow to suitably test the correct working of these com-
  ponents.

- Various **content databases** on an as-needed basis. This could include a catalog of object
  wire frames to be used in model-based object-recognition, a database of speech phrases
  needed for speech generation, and a lexicon and grammar for the same purpose, a library
  of pre-defined action plans (for solving a higher-level goal) or arm trajectories (for frequently
  used manipulation operations), a set of example data or training a neural network, and
  similar things. Such databases are usually required by particular software components.

---

\(^1\)Platform is understood here as the set of software components and tools that make up the robot software
application and the environment in which this software runs.
3.3.3 Capability Building Tools

The tools support for the capability building phase include:

- The **component builder** is probably the most important tool, one that is central to the BRICS approach to robot application development. Components are reusable software elements for which a model must be provided. Over time, the robot software components repository should provide a growing set of such reusable components, so that during application development the component builder will be used mainly to select and configure such reusable components. If the required skill or capability is not available in the repository, then a new component must be constructed. If some legacy robotics software implementing the required functionality exists, it must be refactored to conform to BRICS requirements like the BRICS component model (BCM). A similar activity is necessary for integrating new hardware components (already illustrated in Phase 2), where the vendor-supplied device interfaces must be wrapped into a component structure. These latter two activities, and the provision of new elementary skills and capabilities, usually result in creating primitive components. The component builder should support (at least)
  
  – to identify and define the relevant interfaces for the new components
  – to identify and define the data structures needed by these interfaces,
  – to specify the ports the component should have and which interfaces these ports provide or require, and
  – to specify protocols for using these ports.

Composite components make use of primitive and other composite components. Components can be hierarchically composed to realize more complex skills and capabilities. Component composition requires several substeps, which also occur in putting together
the final system and are further discussed in the next phase (see Section 3.4.3). All these activities need to be adequately supported by the component builder tool, which is one of the central activities in WP4.

- The **model-based code generation engine** allows to generate and configure actual source code from models, at least partially. (For functional details that are not modeled on the model level no source code can be generated, of course. This code has to be supplied by software developers using the component builder).

- As outlined before, various non-trivial components need substantial databases or knowledge bases. The acquisition of these databases and knowledge bases often requires significant effort itself, which should be supported by adequate **content acquisition tools**.

- Using the already known **test builder**, a set of unit tests should be supplied for each component. Even when a component is only selected and configured from the components repository, the definition of some tests testing the specific configuration may be useful in addition to taking over the tests that are already supplied by the repository.

- For actually running the tests, a **unit testing engine** should be available. This could be one of the well-known frameworks for unit testing, or a suitable extension thereof which ensure proper integration into the BRIDE tool chain.

- The **simulation engine** is needed if the tests are to be run on an emulated robot only (e.g. for safety reasons) or if the robot hardware is not yet available.

### 3.4 Tools for System Building

#### 3.4.1 System Building Tool Chain

The activities in this phase are illustrated in Figure 3.5:

![Figure 3.5: The tool chain for the BRICS RAP system building phase.](image)

#### 3.4.2 System Building Artifacts

There is only one tangible output of this phase, but it is of course the central one:

- The final **robot application software** is nothing else than a composite component that can be run standalone. Note, that this does not mean that the application could not foresee other components that can be run independently standalone, but the robot application
software component is required to have this property. It will usually start all system activities and perform all high-level operations on the overall system, like switching to different modes, taking care of serious system faults, etc.

### 3.4.3 System Building Tools

The tool support for the system building phase consist mainly of the **component builder** (already discussed in Phase 3), especially its facilities for building composite components. Component composition requires several activities aside of those already described for the component builder, each of which needs to be adequately supported:

- The **component composition tool** supports to identify and define the relevant subcomponents, which are needed for the component to be defined, to impose suitable type and configuration constraints on them, and to define the connections between the subcomponents.

- The **robot control architecture workbench** supports the developer to link component interface functionality to subcomponent functionality, how activities of subcomponents are synchronized and prioritized, to ensure the proper handling of any error and failure states, etc. This support mainly comes in two ways: A **control pattern repository** provides blueprints for typical connections and synchronization mechanisms between subcomponents, which can be simply applied and configured. The robot control architecture workbench will also allow to build new control patterns.

These aspects of component composition could be supported e.g. by providing specific perspectives in the Eclipse IDE.

### 3.5 Tools for Benchmarking

#### 3.5.1 Benchmarking Tool Chain

The activities in this phase are illustrated in Figure 3.6:

![Figure 3.6: The tool chain for the BRICS RAP benchmarking phase.](image-url)
3.5.2 Benchmarking Artifacts

During development we have defined and collected customer acceptance tests and component tests (unit tests) along the way. The goal of the benchmarking phase is to perform even more thorough tests and assess various system properties by performing specific tests. These tests are the tangible outputs produced in this phase:

- The **stress tests** check the valid operation of a component both in normal as well as difficult or extreme operating conditions; e.g. very high system load, unusual inputs, etc.

- The **safety and security tests** assess component/system behavior by provoking failure situations and attempts to abuse the component/system.

- The **reliability and durability tests** are tests running over longer periods of time, often under extreme operating conditions (e.g. temperature in the operating environment).

- The **performance tests** assess classical performance criteria like maximum operating speed or response times, usage of memory, etc.

Such tests should ideally be performed not only on the application system level, but for each functional component.

3.5.3 Benchmarking Tools

The tools support for the benchmarking phase includes:

- A **test builder**, which allows to define all the mentioned kinds of tests. This tool is actually used in several phases throughout the development process. The test builder can be supported by a **test pattern repository** containing examples and blueprints for tests that can be adapted and configured for the situation on hand.

It may be useful to integrate the various testing tools and facilities into a benchmarking workbench eventually, but this is beyond the realm of this project.

3.6 Tools for Deployment

3.6.1 Deployment Tool Chain

The activities in this phase are illustrated in Figure 3.7:

![Figure 3.7: The tool chain for the BRICS RAP capability building phase.](image-url)
### 3.6.2 Deployment Artifacts

The tangible outputs produced in this phase should be:

- A **target platform**, which includes a revised version of the original hardware platform model, and implied by these changes, the adoption of all the artifacts produced from that original hardware platform model.

- A **runtime architecture model**, which describes how the software elements are to be allocated on the computational architecture at runtime.

- A **maintenance instrumentation model**, which describes the modification or addition of elements for recording various log information and obtaining operating statistics that are helpful during maintenance later on.

### 3.6.3 Deployment Tools

The tools support for the deployment phase include:

- The identification, configuration, and generation of the target platform requires no extra tools, but can be performed with the already available tools of the tool chain. It usually starts with generating a variant of the original hardware platform model, then revising the complete software architecture based on the changes implied by the hardware variant. Even if no such changes occur, the software system architecture may need revisions in order to remove functionality that was added and used during development only.

- The **runtime architecture builder** allows to define a model for mapping software elements (components) to the computational architecture in a runtime architecture model. Many of the design decisions represented in this model are already implied by constraints in the mechanical, electrical, or computational model, but for many software components we may still decide whether or not to run them in their own process or thread and on which computational device. One of the goals of this step is to obtain an allocation that minimizes communication load while achieving good load balancing.

- The **maintenance instrumentation facility** should enable the developer to quickly instrument the application system with functionality for logging various information that is useful or required for maintenance later on, e.g. error and failure states, usage times and frequencies, etc.

### 3.7 Tools for Maintenance

At this point of the project, it is not clear yet whether there will be any specific tools supporting maintenance, aside of all the tools available during development and the instrumentations foreseen during deployment.
Chapter 4

BRASA: The Generic BRICS Robot Application Software Architecture

The past two chapters provided an in-depth look at the robot application development process and a tool chain for supporting this process. The stage is now set for providing insight into the architecture and organization of the software that is the expected outcome of the development process. The BRICS Robot Application Software Architecture (BRASA) is a set of recommendations for the software design of a robot application.

The first section briefly reflects on the notion of “architecture” in robotics and tries to clarify some concepts. The second section introduces a categorization of architectures, where a full robot system architecture is defined to consist of six architectural aspects. Three of those will be considered as software-related and are discussed in more detail in the three subsequent sections. The final section of this chapter provides specifications for the organization of the software.

4.1 On Architectures in Robotics

Various research activities in the past few years\(^1\) have made evident the fact that the robotics community currently cannot make any widely agreed upon statements on the “architecture” topic. While the architecture of a robot had often been an integral part of a scientific publication until the late 80s, the strong focus on methods in the last two decades has almost wiped out the architectural debate. Whether it is a result of this negligence of architecture or not, it remains a fact that practically every robot application developed in the past 20 years adopts a different architecture. This might still be acceptable, if each architecture would have been designed for a significantly different application. This is unfortunately not the case. Currently, if 33 developers would be asked to implement a particular application, chances are high that they would design 33 different architectures. So, we may coin another phrase: “33 architectures for 1 application.” This situation makes application development in robotics time-consuming and expensive.

The architecture debate has been helped by a considerable confusion about the notion of architecture (see e.g. [122], [3], [6], [7], [13], [21], [25], [33], [103], [34], [41], [45], [46], [49], [55], [56], [59], [63], [67], [72], [73], [77], [93], [96], [106], [107], [112], [124], [130], [134], [137]). In order to avoid such a confusion within BRICS, we will clarify this notion and define six different architectural aspects that we could identify in typical non-trivial robot application systems (see Section 4.2). An essential distinction that will be made is between a functional architecture and a software component architecture. Control is a central aspect on both levels. In practice, these different

---

\(^1\)See e.g. various workshops organized by the RoSta project [122] or the SDIR workshop series by Davide Brugali.
aspects are often not separated. The supposedly “simpler” architecture, however, often makes things actually more complex and easily leads to confusion.

We believe that making this distinction is absolutely necessary in order to make progress and speed up robot application development. The situation can be compared e.g. with GUI programming. When the first GUI programming environments arose, they often differed significantly, and programmers applied widely different software designs to build applications. Many of those were riddled by stability problems and frequent failures. Nowadays, there is a more or less established set of design rules for GUI-based applications — not only for the layout of graphical user interface elements, but also for the program design behind it — which make the design of such GUI-based applications much more coherent. If these design rules are followed, a developer well-versed in these techniques can quickly understand an application developed by others. Thus, these design rules are independent of the functionality of the target software system.

The robotics community should target at a similar level of functional independence, i.e. the software architecture, its implementation in source code, and the organization and management of this source code needs to much more independent of the functionality of the robot application, and needs to adhere much more to commonly agreed-upon, (domain-specific) design guidelines.

### 4.2 Architectural Aspects

For a complex service robot, we distinguish six aspects of architecture (see Figure 4.1). The first three aspects are related to hardware aspects, and together can be viewed as the hardware architecture:

- **The mechanical architecture aspect** describes the robot system in all its mechanical aspects. This should include CAD models of each component, how and where they are connected to each other during assembly, and various information about the mechanical components or the overall system that may be of relevance for software development. For example, the color of system parts is of interest to detect when robot parts get into the view field of the robot’s own perception system. The weight of components and of the overall system, and frictional coefficients for the wheels may be of interest when computing the dynamics. And coordinate transformations, for example for data from laser range finders into a robot-centric coordinate system, could be automatically derived when knowing precisely how the sensor system is mounted on the robot. Needless to say, that the information of the mechanical architecture aspect is usually part of the documentation provided for technicians who are to perform mechanical maintenance operations on the robot.

- **The electrical architecture aspect** provides information on all electrical issues of the robot system. It must include all system electromechanical (e.g. actuators), electrical (batteries, switches), electronic (sensors, circuit boards), and computational (microcontrollers, embedded PCs, laptops, etc) elements, and all the wiring (power supply, buses, network lines) between them. This information should be acquired along the way during the actual configuration of the hardware platform. Based on this information for a particular hardware configuration, it is possible to decide whether or not an additional device can be added (e.g. if a sensor needs a USB 2.0 connection, then a free USB 2.0 slot must be available), or it can help to debug difficult problems.

- **The computational architecture aspect** provides information on the computational devices, e.g. which CPUs and GPUs they feature, how much RAM is available, which operating systems they run, and how the computational devices are networked together. For a small robot with a single embedded PC running all application software this aspect may seem trivial. But for a robot featuring several computational devices, this aspect
should contain all information of relevance for mapping a large distributed software system onto this computational architecture and getting it to run smoothly.

The three aspects of the hardware architecture are determined in the platform building phase. The remaining three aspects are related to software and can be jointly viewed as the software architecture:

- The **functional architecture aspect** focuses on functionality rather than software issues. It should identify major functional components, like speech recognition, manipulator control, path planning, etc. and how they interact in order to solve certain tasks.

- The **component architecture aspect** concerns all aspects of the actual software implementation of the functional architecture, especially software modules and their interaction, the interfaces of the software modules and the relevant data structures. As we use component-based programming, components will serve as the predominant concept for modularization. Composite components allow for hierarchical composition of more complex components.

- Finally, the **runtime architecture aspect** maps the software architecture onto a particular computational architecture, mainly by mapping software components onto processes and threads, and by mapping processes and threads onto the computational devices available.

More details are provided in the subsequent sections. The mindful reader may wonder why the notion of control is not appearing in the above discussion; after all, “robot control architecture” is a word frequently used in many robotics papers. The main reason is that control is an integral part of each of the three software-related architecture aspects.

### 4.3 The Functional Architecture Aspect

An overall functional architecture [24] for a robot application that is supposed to cover a wide variety of tasks is very difficult to design from scratch. A stepwise approach may therefore be applied as follows:

1. If not yet done, model every user-level task as a *use case* [100] [36] [10].

![Figure 4.1: The architecture aspects used in BRICS.](image-url)
2. For every use case, identify a small number (≤ 10) of top-level functional modules, which are needed to perform the task. Model these functional modules and define their interactions, e.g. using UML communication diagrams or some other UML interaction diagram [135] [84] [136].

3. Annotate each communication diagram with control information where appropriate.

4. Where appropriate, complement high-level functional modules by refined models providing more detail.

5. Define a functional architecture for robot vitality. Issues to be considered here include:
   - maintaining battery charge levels to ensure safe operation and attending charge stations when necessary,
   - maintaining actuality and a level of constrained completeness of the robot’s internal environment models (exploration and mapping),
   - maintaining location awareness and (dynamic) environmental constraints (localization, obstacle avoidance, acoustic and visual warning signals),
   - ensuring task agenda constraints, including those imposed by other vitality factors (limits on acceptance of jobs),
   - performing (semi)automatic recalibrations when appropriate
   - using models for wear-out of parts to influence control of the robot, when appropriate (wear-out of sensors, actuators), and
   - building hardware fault models, procedures to check for hardware faults, and their scheduling in system operations.

Note that many of these issues are normally required by a fully functional robot application, but are not arising in user-level task descriptions. Thus, they should be considered explicitly by the developers.

6. Integrate the different functional architectures developed for each use case into a single overall system architecture.

The last step is usually the most problematic one, and attempts to do it in a single step rarely yields satisfactory results. A more promising approach may be to integrate only two (or a small number of similar) use cases in a first step. Note that the integration step may induce modifications to the architecture in order to obtain a good design. Especially the control aspects could warrant such modifications. Subsequent steps will then integrate further use cases, until a fully integrated design for the functional architecture is obtained. As the full functional architecture may be quite complex and difficult to understand for people not involved in developing it, it would be a good idea to keep the functional models for all use cases as well as the intermediate models leading to the integrated functional architecture.

The functional architecture aspect is model-level information only. We suggest to rely on various UML 2.x diagrams and models, such as use case diagrams and communication diagrams on this aspect [100] [36] [10] [135] [84] [136].

4.4 The Component Architecture Aspect

The component architecture aspect is about the implementation of the functional architecture in real software. This aspect concerns both model-level and code-level information.
In order to further structure the software architecture, we first survey the different system abstraction aspects used in BRICS. Then the BRICS component model is presented. A base repertoire of components, which are useful for application building, is identified and described thereafter.

### 4.4.1 System Abstraction Layers in BRICS

A standard mechanism developed in computer science to deal with very complex systems is to structure them into layers of abstract machines. This can be applied to robot application systems as well. The abstraction layers we use in BRICS are depicted in Figure 4.2.

![Figure 4.2: The different system abstraction layers used in BRICS.](image)

The abstraction layers are briefly explained from bottom to top:

**Hardware Elements:** On this level, we have actual hardware components such as sensors and actuators.

**Hardware Device Interfaces:** The vendor-supplied software to use a hardware component is on this level. Here we face heterogeneity of hardware interfaces.

**Legacy Algorithms:** This layer contains arbitrary legacy code relevant for robotics, e.g. computer vision or SLAM algorithms, and is considered on the same level as the hardware device interfaces.

**Object-Oriented Device Layer:** This layer provides object-oriented wrapper classes for vendor-specific interfaces. Class interface design follows well-defined guidelines. Harmonization is fostered through class abstraction hierarchies.
Refactored Object-Oriented Algorithms: In order to turn legacy robotics code into reusable components, the algorithms should first be refactored following the BRICS design guidelines for object-oriented design. Note, that the refactoring usually is done such that the legacy code is not being used any more (therefore, the different graphical illustration).

Network-Transparent Services: On the previous two layers, we merely define object-oriented class interfaces. Code using these interfaces must usually run on the same computer and the same process as the low-level interface code. This layer makes the interface usable in other processes and computers, by wrapping the interface into a network-transparent service using communication middleware.

Base Component Layer: Reusability is fostered on this layer by wrapping network-transparent services into base components.

Composite Component Layer: The base components can be combined to implement specific methods and functionality for robotics, e.g. SLAM, path planning, computer vision, etc. Note that the figure shows only two levels of components for this layer, but in fact there could be an arbitrary number of layers here.

Robot Application Layer: Finally, applications can be put together by combining various high-level composite components. The elements on this layer are just composite components, but must meet certain criteria, e.g. being independently executable, or similar.

Components obviously play an essential role in this approach, and they help to meet the different interests of the stakeholders involved in robot application development. Providers of hardware devices should provide reusable components. The abstraction hierarchy provides a clear path towards that goal, via the object-oriented device interface and network-transparent services layers. In a similar fashion, algorithmic libraries need to be refactored and packaged as reusable components. Researchers focusing on solving particular tasks or providing algorithmic solutions for more complex functionalities work on the composite component layer and combine reusable components into more complex, composite components. Application developers and customers collaborate on the application layer to compose target applications, ideally by just selecting, composing, and configuring existing reusable components.

4.4.2 The BRICS Component Model

The BRICS Component Model (BCM) describes and motivates the primitives which are available to develop component-oriented robotic systems. Thereby, the BCM plays a crucial role in the whole project, because it defines the primitives which are used to implement the following (exemplified) use cases:

- a laser scanner sends laser scan data to a map builder
- an algorithm needs to be executed in sequence with another algorithm
- an algorithms parameters are to be tuned by an operator
- when a new tool is mounted on a robot, the kinematics are made aware of this
- and more

In order to realize these use cases the BCM includes the following primitives:
System: A system can be defined as the composition of a set of interacting and interdependent entities. Note, that an entity refers to both components and the connections between them, which represent interactions and inter-dependencies among the components. The BCM allows to describe arbitrary complex systems in terms of the amount of involved components and their organization.

Component: Components are the system constituents providing functionality. There are almost as many definitions of the notion of a component as there are papers about component-based design and development. From a software life-cycle point of view, a component can be a modeling block/class represented in UML in the design phase, a function in the form of C source code during implementation phase, and a running process in the deployment phase. In the context of robotics software frameworks, a component often represents an encapsulation of robot functionality (e.g. access to a hardware device, a simulation tool, a functional library) which helps to introduce structure. Other possible responsibilities include achieving code-level or framework-level interoperability and re-usability, and being composed with other software by third parties. In our work, a component is represented as a block (black box) which defines the boundaries of particular functionality the robot provides. A component can consist of many fine-grained primitives such as classes, functions, etc. In general, the BCM should allow to describe components for arbitrary purposes as the integration of legacy code (e.g. device drivers) or the development of new components (e.g. a novel localization algorithm).

Port: Components need to interact with other components in their environment. The primitives making this interaction possible include ports, interfaces, data types, and connections. The latter three will be discussed below. A port is the software equivalent to the concept of a connector in hardware, and are a component’s communication end-points for its connections to other components. Ports play an important role in component-based design. While in object-oriented programming a class usually provides a single public interface, which can actually be used by any entity that obtains an object reference to an instance of the class, the use of ports allows developers to provide several functionally different interfaces and to constrain their use to well-defined entities that will be connected to a port (see Connections below).

Ports can be typed. The port type may impose constraints on which type of connection may be associated with it. For example, the connection may be required to use particular communication protocols or synchronization mechanisms. Two types of ports are frequently needed in robotics:

- **Data flow port:** A data flow port is used in situations where there is single supplier providing data in regular intervals to one or more consumers. An example is a component which encapsulates a laser scanner device and sends laser scans every 20 msec to anyone connected to it via such a data flow port. Syntactically, the port has a name (e.g. `scan2D`, `position2D`) and an interface for reading and writing data. Via this interface, the port can only communicate information with data semantics to and from other components’ ports; the interaction is supposed to not directly influence control flow on both the sender and the receiver side, and mechanisms for synchronization or advanced handling of communication errors are not foreseen.

- **Service port:** Although important for robotics, data flow ports are not sufficient to build sophisticated robot control architectures. For instance, modifying a component’s configuration or coordinating its activity via a data flow port would be difficult, require extra effort, and lead to suboptimal designs. Therefore, a component model should
feature a port type with control flow semantics. Syntactically, the port has a name and an interface made up of a collection of methods or functions, referred to as services.

In addition to the difference in information semantics, data flow ports and service ports are usually associated with different interaction patterns. While service ports usually imply synchronous interaction between components with clearly assignable client and server roles, data flow ports usually imply asynchronous interaction between components with clearly identifiable publisher and subscriber roles.

**Interface:** An interface is a set of operations made available to the outside by a software entity. An interface is usually defined by a set of method signatures.

**Data types:** The classification of data which is communicated between components of the system is done through data types. Both the arguments and return values for the functions/methods specified in the interfaces used in component ports need to be agreed upon in order to ensure correct representation and interpretation of the communicated data, especially if the two connected components eventually reside on different computers running different operating systems, and are implemented in different programming languages. Automatic translation or conversion of data types across languages and systems can be difficult or even impossible if incompatible data types are used. Interoperability can be fostered by providing a standardized library of domain-specific data types, which should be designed to minimize or avoid such incompatibilities.

**Connection:** Connections provide the actual wiring between ports of different components. That is, while a port is a component-level mechanism to make a particular component interface available to the outside, connections perform the linking between ports. With this role, connections are the concept suitable to encapsulate any details about communication protocols and synchronization. This is in line with the definition in [82], where connections mediate interactions among components. That is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required. From an implementation perspective, connections may be realized as simple as memory access or a UNIX pipe, or as sophisticated as TAO [105], ICE [140], ZeroMQ [141] middleware runtimes and their respective interaction patterns. For instance, publisher/subscriber, client/server, peer-to-peer are most common interaction patterns. From a modeling perspective, a connection is a directed link connecting two ports.

### 4.4.3 A Base Repertoire of Components

The composition of components from simpler components naturally provides a hierarchical organization of all components of a robot application. In fact, the application itself is nothing more than a complex composite component, which may fulfill some additional requirements, like meeting certain completeness constraints.

Aside of the hierarchical view of components induced by the composition mechanism, we can further classify components into a set of different categories as follows:

- basic hardware service components, like sensors and actuators
- background functional service components, like SLAM, ROI detection, noise detection, security services
- foreground functional service components, like task planning, people recognition, speech recognition
- user-level or task-level service components
• operational control components
• system vitality services
4.5 The Runtime Architecture Layer

The runtime architecture layer concerns the mapping of the software components onto processes and threads running on the available computational devices. If the target application is very simple and features only a single embedded computational device to which all sensor and actuators are connected, this mapping is quite simple, of course. More complex robot systems, like those used in projects like DESIRE [40], RobotCub [121], and RoboCup@Home [128], for example, are equipped with multiple computational devices, and mapping the software components to them may be non-trivial and have significant influence on system performance and even system safety. It is quite questionable, whether the co-allocation of software components that have to meet realtime constraints in order to ensure system safety and software components that have large and hard-to-predict resource requirements (in space and time) on the same single-core PC is really a clever idea. For these non-trivial cases we assume that the robot hardware platform feature a set of different computational (programmable) devices, which are networked together with some standard network technology, e.g. Ethernet, to form some kind of local area network. Each of these devices is assumed to be independently bootable. At least one of them should have persistent memory (a hard disk or FLASH memory) to store executables, runtime libraries, configuration files, necessary databases, log files, and whatever other information may be required during actual operation of the robot application.

The model-level information of the runtime architecture layer can be captured in UML 2.x deployment diagrams. The implementation-level artifacts to be automatically derived or manually constructed include:

- the allocation of files on persistent memory of particular computational devices,
- procedures for controlled system startup and shutdown,
- the configuration of automated service initiation,
- and the instrumentation for monitoring and logging.

4.6 Software Organization

So far we were mostly concerned with models and other abstract information. Eventually, the models need to be met by implementations. The question arises how models, source code, executables, configuration files, third-party software, and other information should be organized in a file hierarchy. This is a non-trivial and important question, as this organization often impacts the ease with which people can understand a system and take over further development and maintenance tasks. In BRICS, we propose to follow a simplified approach:

- We keep the file hierarchy as flat and narrow as possible, but as deep and wide as it makes sense and can be justified by clear structuring rules.
- We maintain meta information, stored on a per file or per directory basis, for keeping additional information on elements of the file hierarchy.
- We exploit this meta information to build additional virtual file hierarchies, which may be arbitrarily deep, wide, and connected as may deem appropriate.

A tree structure for the file hierarchy is assumed, which may be complemented by the possibility of adding symbolic links. The proposed structure for the file tree is depicted in Figure 4.3; further explanations are provided below.

2The often widely different ways in which large software projects are organized shows that finding a good file hierarchy is not simple. Five different developers often come up with five different ways to organize the code.
Figure 4.3: BRICS standard for the organization of robot application software in a file hierarchy. If a directory tag is followed by a colored triangle, then a copy of the tree with the same color in its root circle is placed under this directory. The fully expanded tree under the red circle holds a complete robot application. Items in angle brackets are placeholders for an arbitrary number of instances. Various leaf directories in the resulting file hierarchy may contain further subdirectories as appropriate.

```
./models/  holds all model information.
./models/mechanical/  holds models describing the mechanical architecture. May contain subdirectories for particular components.
./models/electrical/  holds models describing the electrical architecture.
./models/computational/  holds models describing the computational architecture. May contain subdirectories for particular components.
./models/runtimeArch/  holds runtime architecture models.
./models/componentArch/  holds models describing the component structure of the robot application. May contain subdirectories for subcomponents. Directories in this tree struc-
ture contain mainly symbolic links to actual model files, which are kept in the models directories of the components.

./models/functionalArch/ holds models describing the functional architecture of the robot application.

./models/simulation/ holds models and information relevant for simulations.

./models/simulation/simulationObjects/ holds models for objects to be simulated. This includes all classes of objects needed to model the environments in which the robots are to operate, all objects that need to be perceived and/or manipulated by the robot, or are otherwise task-relevant.

./models/simulation/emulationObjects/ holds models that allow the simulation engine to emulate the robot hardware.

./models/simulation/experiment/ holds information pertaining to the execution of simulation experiments.

./components/ holds all information about components.

./components/primitive/ is the subdirectory that holds information on primitive\(^3\) components. Usually component wrappers around network-transparent services.

./components/composite/ holds components composed from primitive of other composite components.

./components/application/ holds composite components that can be run as standalone application. Usually holds only a single composite component that makes up the target robot application.

./nts/ holds the network-transparent services. Should contain a subdirectory for each object-oriented device interface.

./oods/ holds the object-oriented device interfaces. Should contain one subdirectory for each device driver class in ./external/deviceDrivers. Also all classes resulting from hardware device abstraction.

./interfaces/ holds interface definitions.

./datatypes/ holds data type definitions.

./external/ holds all external software packages and code.

./external/deviceDrivers/ holds the vendor-specific device drivers for sensors and actuators.

./external/middleware/ holds the communication middleware to be used.

./external/frameworks/ holds any robot programming frameworks used.

./external/libraries/ hold any other kind of external library code used.

./tests/ holds (application) system-level test cases, e.g. the customer acceptance tests.

./logs/ holds system-level log information

\(^3\)Primitive in the sense of non-composite.
.//tools/ holds all kinds of tools to operate and maintain the robot application.

.//documentation/ holds all documentation

.//user/ holds executables for the end-user. This could possibly be empty, or contain one or a small number of executables that get the robot into service, control it during operations, or shutting it down.

The sub-structure applying to each primitive, composite, and application-level components as well as to network-transparent services and object-oriented device interfaces is as follows:

.//components/composite/models holds any model information.

.//components/composite/source holds the actual source code. This directory may contain symbolic links to the directories holding the sub-components of the component at hand.

.//components/composite/configuration holds configuration files.

.//components/composite/binary holds executable binaries, if any.

.//components/composite/tests holds test cases.

.//components/composite/logs holds various log files.

.//components/composite/tools holds tools for managing the component.

.//components/composite/documentation holds whatever documentation is available. Should contain the javadoc/doxygen documentation of the source code in a subdirectory.

.//components/composite/user holds any programs and tools to be used by an end-user. Empty for most components.

As a last remark we want to emphasize that this structure is the result of a best effort in integrating hindsight and foresight, based on our experiences and discussions with colleagues and practitioners. Nevertheless, the structure could be and will be adapted if necessary.
Chapter 5

BROCRE: The BRICS Open Code Repository

5.1 The Role of BROCRE in BRICS RAP

The BRICS Open Source Repository (BROCRE) is a repository of software available for building robot applications. It is an essential ingredient for supporting robotics software re-use in BRICS. It helps to decouple activities of different stakeholders, as e.g. hardware component providers can focus on delivering high-quality, easy-to-reuse software interfaces with their components; they can also test them in application contexts, if BROCRE contains examples of fully worked out robot applications (even if only run in simulation). BROCRE helps to foster more rapid dissemination of technological advances, bug fixes, and software updates, and it could pave the way for future standards in robotics, if developers start to reuse interface definitions and data structures already provided by BROCRE.

5.2 BROCRE Structure and Dependencies

We propose to maintain and exploit a structural similarity between

1. the BRICS Robotics Open Code Repository (BROCRE),
2. a local projection of BROCRE,
3. the actual application software system under development, and
4. the deployment version of this application.

This structure is illustrated in Figure 5.1 and further detailed below.

BROCRE is a globally accessible code repository; it may be stored and maintained in a distributed fashion, but from a logical perspective it can be viewed as a centralized repository. The structure of BROCRE is suggested to have eight major divisions:

- **DataTypes** contain data type definitions which are used in interface definitions.
- **Interfaces** contain interface definitions, usually as definition of abstract classes.
- **Object-oriented Device Units** provide object-oriented class wrappers for all kinds of vendor-specific, low-level device interfaces or are refactored best practice algorithms provided as object-oriented classes.
Network-transparent Services turn functionality provided by object-oriented device units into services that can be transparently accessed via networks.

Components should be the smallest granule of release, if the developers adopt a component-based programming approach. Both elementary and composite components will be stored here in a flat directory. The component names used here will have to be fully qualified names and usually contain name space information in order to avoid conflicts between different developers providing different components under the same component name.

Applications can contain both specific robot software applications that can run only a possibly highly customized platform, and patterned applications, which have been abstracted from real applications in a generalization process.

Distributions are just definitions of sets of components and applications. They can be used as a convenience mechanisms to identify a subset of BROCRE models and code under a single name. Example used would be distributions for educational showcases based on the youBot platform, domain-specific distributions like for manufacturing logistics, or vendor-specific distributions, e.g. a distribution suggested and partially or fully supported by KUKA.

External Software Units contains all external software packages, or appropriate information to access, download, and install it from another repository, such as vendor-specific device drivers, middleware packages like TAO [105], or ICE [140], robot software frameworks such as ROS [48] [91] [97], OROCOS [118] [28], or OpenRTM [123] [4], and libraries such as boost [116], etc.

Aside of the BROCRE repository, the download domain developed by BRICS would have to support the download of BRIDE, of showcases, and supportive documents such as tutorials, documentation, and similar information.

The local projection of BROCRE contains all BROCRE elements which are either already known to be required for the application (e.g. software components for the hardware devices as determined in the platform building phase) or qualify as potentially reusable components. As an example for the latter, assume we have configured a hardware platform without a laser scanner, but an omnidirectional camera. Components implementing SLAM algorithms which require a laser scanner cannot be applied in our application, but those implementing some visual SLAM algorithm are possible of use for implementing the application.

The application software system under development tree contains those components that are actually used in the current version of the application software. It consists of elements originating from BROCRE, elements locally added in various stages of the development process, e.g. models, configurations, locally provided source code, and elements that have been automatically generated by tools, e.g. the source code derivable from component models or code for test cases.

The deployment version is further constrained to contain only those parts that will go onto the target platform. The patterned version of an application, if available, is the attempt to generalize the application code in order to make it more reusable.

The proposed structure for the file tree of each of the above is depicted in Figure 5.1, where under each directory followed by a red triable a complete copy of the tree shown in Figure 4.3 is placed. Further explanations are provided below.

<bricsDomain>/brocre The logical root of the BROCRE repository.
<bricsDomain>/brocre/datatypes Directory holding data type definitions.
<bricsDomain>/brocre/interfaces Directory holding interface definitions.
Figure 5.1: Organization of BRICS software in a file hierarchy.

**<bricsDomain>/brocre/oods** Directory holding object-oriented devices.

**<bricsDomain>/brocre/oods/**<oodName> Directory for a particular object-oriented device.

**<bricsDomain>/brocre/nts** Directory holding network-transparent services.

**<bricsDomain>/brocre/nts/**<ntsName> Directory for a particular network-transparent service.

**<bricsDomain>/brocre/components** Directory holding components.

**<bricsDomain>/brocre/components/**<compName> Directory for a particular component.

**<bricsDomain>/brocre/applications** Directory holding applications.

**<bricsDomain>/brocre/applications/**<appName> Directory holding a particular application.

**<bricsDomain>/brocre/distributions** Directory holding distributions.

**<bricsDomain>/brocre/districutions/**<distName> Directory holding (meta)files defining a particular distribution.

**<bricsDomain>/bride** The logical root of the BRIDE development tools.
<bricsDomain>/showcases A central repository of showcases. Showcases are exemplary applications that demonstrate the use of BRICS technology.

<bricsDomain>/showcases/industry contains showcases relevant for industry.

<bricsDomain>/showcases/research contains showcases from research.

<bricsDomain>/showcases/education contains showcases for education.

<developerDomain>/brocreSelection contains a local copy of a user-defined subset of the BROCRE repository, which developers decide to have a local copy of. This makes sense mainly if the developers develop and/or maintain several robot applications, and serves as a locally shared place for components used in several application projects.

<developerDomain>/brideSelection contains a local copy of a user-defined subset of the BRIDE tool chain.

<developerDomain>/showcasesSelection contains local copies of a user-defined subset of BRICS showcases.

<developerDomain>/<appname> is the local root directory for a robot application software project.

<developerDomain>/<appname>/brocreProjection contains local copies of the BROCRE elements required by the application software.

<developerDomain>/<appname>/develop contains the actual robot application in its development version. This is the main working tree for the software developers.

<developerDomain>/<appname>/deployed contains the deployment version of the application on the developer site.

<developerDomain>/<appname>/patterned contains the abstracted (patterned) version of the application on the developer site.

<deploymentDomain>/<appname>/deployed contains the robot application software on the actual target platform hardware.

5.3 BROCRE Software Maturity Model

When software repositories grow sufficiently large, both their maintenance and their use can become very difficult. If we imagine a professional software developer at a robot systems integrator who is just starting to get acquainted as a user with a large software repository, this user would have several desirables:

**Granularity:** The granularity of the repository items should be "right". It is, of course, hard to define what "right" is. Granularity is too small, if the user needs to retrieve many items from the repository in order to do something reasonable with it. Granularity is too large, if the user is forced to retrieve large items from the repository, but uses only small parts of it. There is a lot of leverage in between.

**Documentation:** Every item in the repository should have a concise and clear description of the functionality that it provides.
Classification: As soon as a repository grows sufficiently large, the user must be given additional structure, usually in the form of introducing different software "categories", e.g. "development tools", "programming environments", "business software", etc. The categorization may be flat or hierarchical. If sufficiently large, categorizations grow similar to ontologies.

Dependencies: The user needs to see what other entities this item depends on. This holds especially if the software item depends on particular piece of hardware, such as a highly specialized sensor. It is also of interest to know which other items depend on the item on hand.

Maturity: The user needs to know how stable the software is. While academic researchers may be highly interested in items that are at the forefront of the state of the art but still very experimental, an industrial customer may be interested only in really mature software items.

Activity: The user also needs to know if the software is actively developed and maintained, how the development and maintenance team is composed and how they are organized.

Distribution: Information about how often an item has been requested and how well it is distributed in the community may also influence the developer’s decision to consider the software item.

Usage: Information about the actual usage of the repository item. This is really hard to come by information, but many users consider it critical. Many software repositories sell distribution information as usage information.

Subsequently, we describe the specifications we suggest for the BRICS project:

- For the BROCRE repository, we suggest three levels of granularity for repository items:

  1. Components are considered the smallest reusable elements. As composite components already depend on other components, they provide a mechanism to package arbitrarily large and complex functionality into a single reusable element.

  2. Applications are also made reusable elements. Aside of the inclusion of full applications, which may not be the most viable option for robot applications of commercial end users, the inclusion of a well-defined set of application meta-information should be supported. Such meta-information may, for example, contain only information about the usage of components provided by BROCRE, but not on components developed by the customer or a system integrator. No information on the internal structure of the application needs to be revealed except for which publicly available, open source components are internally used. The benefit received in exchange is that the repository can gather more accurate data about the actual use of components, which is likely to influence decisions about improvements, maintenance, and support.

  3. Distributions are simply sets of components and applications. As outlined previously, distributions provide a simple mechanism to define domain-specific and vendor-specific granules of reuse.

- Each unit in the repository must carry an appropriate description of its purpose, which will be reviewed and has to be accepted by a BROCRE maintenance team before a unit can be included in BROCRE.
• Finding a good classification to classify contributions to an arbitrarily large repository of robotic software is very difficult. Although we will provide an initial classification, based possibly on classifications used in the organization of major robotics conferences, we plan to provide a facility where users of BROCRE can provide additional, even different classifications.

• The meta-information for a unit must maintain a single list of units upon which the unit on hand depends directly. Given this information for all units of the repository as a database, the following information can then be automatically inferred by appropriate database queries:

1. The closure of all units upon which the unit on hand depends either directly or indirectly.
2. A list of units which directly depend on the unit on hand.
3. The closure of all units which depend either directly or indirectly on the unit on hand.

• The maturity of a BROCRE unit will be described according to the state graph depicted in 5.2, which is inspired by the maturity model of the Debian Linux distribution. The states and activities are briefly detailed as follows:

contributed is the state of a third-party BROCRE unit when it is initially submitted to BROCRE. Developers can make arbitrary changes to it until they initiate the promotion process. The BROCRE repository management team will decide about promoting the unit to experimental status. Otherwise, the unit will be removed from the repository. Units should not remain very long in this state, preferably less than 2 to 4 weeks.

experimental is the state of units that are considered to be eventually of value for robotics developers, but cannot yet be considered as part of distributions. Arbitrary changes to functionality and interfaces may still occur. As long as interfaces are not stable, developing against such interfaces is risky and may incur cumbersome code adaptations later on. A unit may stay for some time in experimental state. No detailed development plan needs to exist.

unstable is the state to which a unit gets promoted from experimental state when it is considered a candidate for a distribution. As distributions should contain only stable units, a more formal process to get units into this state needs to be applied. This process starts with promotion to unstable state. Contrary to being in experimental state, a unit in unstable state needs to have objectives, a work plan, and a schedule. Objectives can relate to functionality, performance, or quality criteria. The work plan should foresee major development steps and milestones, the schedule needs to determine deadlines for milestones and assign periods to major development steps. Interfaces and functionality may still change arbitrarily when in unstable state, but usually converge towards the objectives. Testing is usually done as part of test-driven development, focuses on functionality and is performed by developers (α-testing). When developers believe the desired functionality is completely implemented, designed well, and sufficiently robust, promotion to next state is initiated.

1It can probably be compared with an attempt to classify music into genres, something on which even web site like Apple’s iStore have more or less dramatically failed, because there exists no widely accepted agreement on such a classification. In music, such classifications are also highly volatile and can change significantly within a few weeks. People in interactive media have therefore are looking into various kinds of user-based tagging systems.
testing is the state in which a unit is released to a limited audience willing to help testing and debugging a unit (β-testing). Functionality usually does not change any more; the focus is on making the unit as robust and performant as possible.

frozen is the state a unit will get if testing is considered completed and the unit is considered being suitable for inclusion in distributions. In frozen state, further tests must be performed which focus on compatibility and interoperability with a set of other units to be included in the distribution. The objective is to avoid having conflicting units in a distribution, or at least to identify and document such conflicts. An example of a conflict is e.g. two units requiring different, incompatible versions of a library which cannot coexist and be used simultaneously. When sufficient testing and debugging has been performed, a unit can be promoted to stable state.

stable is the state in which eventually the majority of repository units need to be. Units in stable state are pieces of software with well-defined functionality and interfaces, completely implemented, well debugged, robust, performant, interoperable, and therefore highly reusable. Distributions should include stable units only. Small bug fixes may
still be applicable to stable units, but anything that requires extensions or modifications of data types, interfaces, or functionality will be collected in form of tickets to be considered for the next major release.

**archived** is the state a unit will get when it gets superceded by a new version. Archived units are still retrievable (like units in any other state), but should not be included in distributions any more.

**deprecated** is the state a unit will get if the developers and maintainers intent to stop supporting the unit. All units including the unit will be notified about this, so that they can start considering alternative units.

**unsupported** is the state a unit gets when developers/maintainers have stopped supporting the unit.

The two dotted lines in Figure 5.2 describe the range of states a unit needs to be in in order to get advertised. Advertising means that a tool which helps to define a distribution by offering lists of available units in various categories will include only units in states *unstable, testing, frozen, and stable*.

- The description of the activity will include a status attribute, which can be set by the developers/maintainers of the unit. Currently, we foresee three possible values: active, maintaining, inactive. Methods to automatically defer activity from developer/repository interaction logs will be investigated and may provide additional information.

- The information on unit distribution basically includes numbers of downloads, possibly separable by geographical region, user community, or other criteria. A new industrial user should, for example, be able to immediately see, which control components are preferred by other industrial users, or by users in a particular domain.

- As outlined previously, information about actual usage of software is hard to get. From a developers’ perspective, an ideal situation would be if every unit could send back information to the repository when it has been successfully installed, every time it is included in code, every time when it runs, etc. For obvious reasons, including privacy and security issues, this currently seems to be completely unrealistic, at least in robotics. Users/developers simply do not want to, or are not allowed to, reveal such an extent of information about their application. On the other hand, *if* they are using components from an open source repository such as BROCRE, they have an interest that these components will be maintained for some foreseeable future, that they get adapted to different operating environments, see bug fixes, and extensions of their functionality. If there is no feedback at all, then repository maintainers have no clue about which components are actively used and should be maintained, and which are rarely used and may not need as much attention. Therefore, we plan to provide a facility which lets developers/users include meta-information about their application in the BROCRE repository. This facility will allow a user to reveal only less sensitive information, like which open source components have been used in the application. Thereby, the user gets a means to publish certain dependencies which she has an interest to be maintained. This way, data for a possible quantitative measure of "best practice" could be developed.

Given this range of aspects, it is clear that the BROCRE repository needs a well-defined set of management tools. These are outlined in the next section.
5.4 BROCRE Management Tools

In order to coherently manage a community-developed, community-maintained software repository for robotics, we suggest to use a tool chain supporting the following activities:

**Distributed source code maintenance and versioning:** Developers of components in BROCRE will be spread out throughout the world. An effective tool to manage source code developed by a physically distributed group of developers is essential. We currently suggest to use `git` for this purpose. An open issue still is whether user-defined version tags will be sufficient to separate user-level versioning from tool-level versioning, which is highly advisable [86].

**Software maturity management:** The maturity model defined above requires to obey various formal steps, especially when promoting a software package from one state to another. This process should be supported by an appropriate tool. One option that can be seriously considered is the use of paper management tools (such as PKP [119]) for online journals (such as JOSER [43]).

**Meta-information management:** Unless the packages or units in a software repository are properly documented and categorized, they become almost useless when the repository is sufficiently large. Categorization must be adaptable by actual users.

**Distribution assembly and management:** Defining and maintaining distributions (in the sense used in BROCRE: sets of software components from BROCRE) can be a tedious task, unless it is supported by appropriate functionality to search BROCRE, manage consistency and completeness of distributions, etc.

**Semi-automated management of downstream and upstream code distribution:** When a BRICS developer has chosen and downloaded a particular subset of BROCRE (in `<developerDomain>/brocreSelection`), then this subset should get automatically updated whenever the main BROCRE repository is updated (downstream code updates). Supporting developers to submit code to the BROCRE repository, being it small updates or extensions to components or publishing a new application, is equally important (upstream code management).

**Issue tracking:** Support for collecting information on bugs and their potential resolution and requests for extension of functionality are standard requirements for open source software repositories.

**User community management:** To create and maintain a lively user community, various supportive measures like maintenance of groups of registered users, topical wikis, and mailing lists are required.

**User interface:** It would be highly desirable to have a unique, integrated interface to access all the above functionality instead of forcing the user to learn how to use several different tools.

It seems that none of the widely available software management tools supports all of the above functionality, but we hope that a small collection of such tools can be set up and integrated sufficiently with the BRICS tool chain to cover this desirable functionality as far as possible.
Chapter 6

Conclusions

6.1 Summary of Current State

The purpose of this deliverable is to provide specifications that guide further development in BRICS, especially on architectures, modules, modularity, and interfaces for the BROCRE software platform and a robot control architecture workbench. Because currently no widely accepted software development process model for robot application development exists, but many interactions and dependencies between the development process and the above topics exist, we included a proposal for a coherent and encompassing software development process model, called BRICS RAP. We further included a discussion of the implications of BRICS RAP on the BRIDE tool chain, then propose a generic robot application software architecture (BRASA), and finally supply specifications for the BROCRE, the BRICS Open Code Repository.

We summarize the major results in the following list:

- A comprehensive software process model for robot application development — BRICS RAP — has been developed. It reflects the needs for more professional software development in robotics and includes numerous steps aimed at assuring that the quality of the developed robot application software meets customer requirements.

- The software development process integrates elements from model-driven development, agile development methods, and component-based programming, all of which are considered state-of-the-art techniques in modern software development.

- For every step in the BRICS RAP process, we have described the desirable associated tool chain, including requirements for the specific tools involved and an informal characterization of the major artifacts required or produced.

- We have identified and described six different architecture aspects relevant for robotics development projects and organized them into a system architecture hierarchy. The three architecture aspects making up the software architecture have been described in detail.

- For the functional architecture aspect, a stepwise procedure for defining it was provided.

- The component architecture aspect (which describes what is commonly/naively associated with the notion of software architecture) has been related to a revised version of the system abstraction layers already previously provided in BRICS.

- A brief summary of the current proposal for the BRICS Component Model has been included.
• A proposal for standardizing how to structure the software, including model information, into a file hierarchy has been developed.

• A structure for organizing the BROCRE repository has been described.

• A proposal for the meta-information to be acquired and maintained for BROCRE entities has been made.

• A software maturity model for BROCRE entities has been defined and described in detail.

With respect to the title of this document, the following conclusive statements apply:

• Specifications on the architecture of robot applications is found in Chapter 4, on the architecture of BROCRE in Chapter 5.

• Specifications on modules and modularity are found in chapters 2 to 5. Chapters 2 and 3 by way of the development process and the artifacts involved in the process, which frequently represent "modules" of the application from a holistic perspective. The main information on modules and modularity is in Chapter 4, which identifies all relevant entities required for a complete, professional robot application software, organizes them in a standardized file hierarchy, and, by virtue of the component-based organization of the main functional modules, provides a generic mechanism to structure arbitrarily large applications in compositional component hierarchies.

• For the specification of interfaces, the BRICS component model provides a sophisticated, generic mechanism.

• Specifications on all the above aspects for the BROCRE repository itself are provided in Chapter 5 and to some extent already in Chapter 4.

• Specifications for the Robot Control Architecture Workbench can currently only be given to a partial extent; revised specifications will be given in a future deliverable. The basic mechanism for implementing robot control architectures will be based on the composite component model. This allows to distribute control horizontally and vertically across the component hierarchy making up an application; on each level the control aspects of a limited number of interacting (sub)components are considered. We currently investigate whether there exist repeating patterns of interaction between components and whether and how these patterns can be identified, described, generalized, and reused. Assuming such patterns can be identified, the robot control architecture workbench simply needs to aid the developer in identifying and applying them.

6.2 Future Work

The comprehensive analysis and design of the robot application development process has revealed the need for a far more elaborate tool chain than foreseen in the initial project plan. For example, most of the tools needed for Scenario Building, Deployment, Maintenance, and to some extent also Benchmarking, need substantial development effort, for which BRICS does not have adequate resources. We will therefore have to carefully select on which parts we need to focus our efforts, and which parts of BRICS we can afford to currently not support fully. Contributions by other research projects are desirable and will be considered appropriately.
Bibliography


