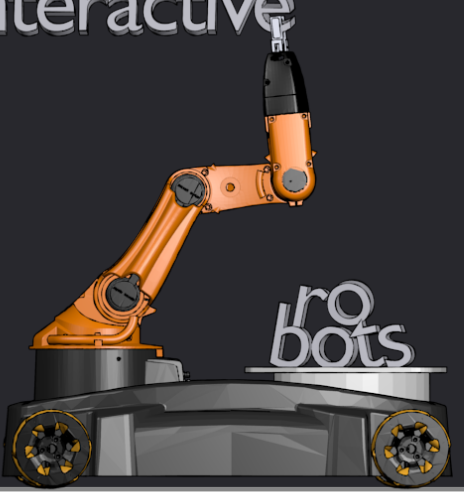
Robust autonomy for interactive



Yury Brodskiy

PROPOSITIONS

belonging to the thesis

Robust autonomy for interactive robots

1. Viewing of the robot control software as a combination of models allows to make design intentions explicit and clear, thus improving quality of the produced software, especially reliability.
2. Component-based software development allows to combine models written in different modeling languages in a single application.
3. Models derived from different meta-models describing the same object, like different witnesses, convey different pieces of information.
4. High-quality motion control software complemented with fault-tolerance algorithms is essential for building autonomous robots.
5. Observing energy flows in the system offers useful inside information about system behavior.
6. For a researcher, finding a correct and precise definition of the problem is the major part of the problem.
7. Good information is hard to find, but it is even harder to use.
8. Stopping is not an adequate response of an autonomous robot for an abnormal event. “Freeze? I'm a robot. I'm not a refrigerator.” – Marvin (Douglas Adams, The Hitchhiker's Guide to the Galaxy)
9. It takes a village to write a book.

Yury Brodskiy 20 February 2014

Robust autonomy for interactive robots

Y. Brodskiy

Faculty of Electrical Engineering Mathematics and Computer Science, University of Twente

Robotics and Mechatronics group

CTIT Ph.D. Thesis Series No. 13-285
Centre for Telematics and Information Technology
P.O. Box 217, 7500 AE
Enschede, The Netherlands.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics).

Title: Robust autonomy for interactive robots

Author: Y. Brodskiy
ISBN: 978-90-365-3620-2
ISSN: 1381-3617 (CTIT Ph.D. Thesis Series No. 13-285)
DOI: 10.3990./1.9789036536202

Printed by Ipskamp in The Netherlands.

ROBUST AUTONOMY FOR INTERACTIVE ROBOTS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 20 Februari 2014 om 12.45 uur

door

Yury Brodskiy
geboren op 15 September 1984
te Sint-Petersburg, Rusland

Dit proefschrift is goedgekeurd door:

prof.dr.ir. S. Stramigioli, promotor
dr.ir. J.F. Broenink, assistent promotor

Graduation committee

Chairman and Secretary

prof.dr.ir. A.J. Mouthaan	University of Twente

Supervisor

prof.dr.ir. S. Stramigioli	University of Twente

Assistant Supervisor

dr.ir. J.F. Broenink	University of Twente

Members

prof.dr.ir. H. Bruyninckx	KU Leuven
prof.dr. R. Babuska	Delft University of Technology
prof.dr.ir. P.J.M. Havinga	University of Twente
prof.dr. D.K.J. Heylen	University of Twente

'Nee heb je, ja kun je krijgen.'
Dutch wisdom

To my wife Edit –
my muse and my critic

Summary

The growing trend in robotics towards applications in an unstructured environment offers new challenges in robotic software and control. Assisting, interacting and serving humans, new robots will literally touch people and their lives. Performing tasks devised for such new robots demands high levels of autonomy, robustness and safety, and the challenge to ensure these qualities has become a task of robot control software. The reliability of robot control software is the cornerstone needed to achieve high levels of autonomy, robustness and safety.

The goal of this research is to study and suggest ways to improve the reliability of a robot, with a focus on its motion control software. Robot motion control ensures the ability to proceed with a designated task, and it is one of the vital parts of a robot, where non-functional requirements such as robustness and reliability are essential for high quality of the overall system. In this work, three threats to reliability and safety of the motion control software were identified and addressed: the quality of software implementation, the external faults from a connection to the physical domain such as failures of a sensor, and the external faults from a connection within the cyber domain such as communication to other components.

A modeling approach to software development is advocated here as the means to improve quality of the produced software. The proposed approach for software development uses 'uniform' modeling of the software components to improve their reliability. The need to model software from different perspectives is emphasised here, with the software practice of separation of concerns used to identify different perspectives from which a software component has to be modeled.

To achieve the 'uniform' coverage of modeling perspectives, a tool chain that combines two different modeling tools was developed. Each tool focuses on a different engineering role. Two tools were used as an example of the approach, BRIDE and 20-sim. BRIDE is used to model the architecture of the system, and 20-sim is used to model the algorithms.

Robustness required for long-term autonomy operation lies beyond high quality of software, it requires algorithms that can tolerate external faults.

The tolerance of external faults originating from a connection of the cyber domain to the physical domain such as failures of a sensor can be addressed using a fault-tolerant control. Failure of hardware is an inevitable event in a long-term autonomous operation; therefore, the ability to detect and react to such failures is essential for an autonomous robot. Fault-tolerant control strategies as presented here allow a mobile manipulator to mask failures of a sensor or motor in a joint or a wheel drive and thus to continue task execution with reduced performance. The presented use case demonstrates that use of fault-tolerant control allows avoiding violent reaction of the robot on sensor failures and partially preserve the control performance.

Coordination of cooperation between multiple robots over a network brings an other

type of external faults, that originates from communication between software components. While network communication widens the range of possible application for robots, it brings a considerable challenge to control interaction with coordination of cooperation over a network, due to communication imperfections. An architectural pattern for control software advocated here allows to tolerate communication failures, thus allowing to build robots that require access to distributed resources and interaction in unstructured environment. The passivity layer, described here, represents a new type of safety element and as such can be combined with any control strategy. An application example demonstrates the gain in robustness with respect to time delays.

High-quality motion control software complimented with fault-tolerance algorithms is essential for building autonomous robots designed for applications in an open environment. Achieving fault-tolerant high-quality motion control software was the goal of this work, and presented combination of the development process and the fault-tolerance algorithms address this goal.

Samenvatting

De stijgende trend in gebruik van robots in ongestructureerde omgevingen geeft nieuwe uitdagingen aan software en regeltechniek van deze robots. Deze nieuwe klasse van robots zullen in hun werk van assisteren en dienen van mensen, letterlijk met deze mensen in aanraking komen. Om deze nieuwe robots hun taken goed uit te laten voeren is een grote mate van zelfstandigheid, robuustheid en veiligheid vereist, wat een grote uitdaging is voor de ontwerpers van robotregelsoftware. De betrouwbaarheid van deze robotregelsoftware is essentieel om deze kwaliteit te bereiken.

Het doel van dit onderzoek is het onderzoeken en suggereren van manieren om de betrouwbaarheid van een robot te vergroten, gefocust op de regelsoftware. De regelsoftware van een robot zorgt ervoor dat de robot de toegewezen taken kan uitvoeren terwijl niet-functionele eisen als robuustheid en betrouwbaarheid noodzakelijk zijn voor de kwaliteit van het systeem in zijn geheel. In dit werk zijn drie bedreigingen voor de betrouwbaarheid en veiligheid van de regelsoftware geïdentificeerd en aangepakt: de kwaliteit van de software-implementatie, externe fouten in de communicatie met het fysieke domein, bijvoorbeeld vanwege een kapotte sensor, en externe fouten in de communicatie binnen het cyberdomein, bijvoorbeeld in de communicatie met andere componenten.

In deze studie wordt een modelgebaseerde benadering van software ontwikkeling aanbevolen als een middel om de kwaliteit van de geproduceerde software te verbeteren. De voorgestelde aanpak maakt gebruik van ‘uniforme’ modelvorming van software componenten om hun betrouwbaarheid te verbeteren. De noodzaak om software vanuit verschillende oogpunten te modelleren wordt benadrukt, en de aanpak van “separation of concerns" is gebruikt om vast te stellen vanuit welke oogpunten de software moet worden gemodelleerd.

Om alle verschillende aspecten op een ‘uniforme’ manier af te dekken is een toolchain samengesteld die twee verschillende tools combineert. De tools richten zich op verschillende ontwerprollen. Twee tools zijn gebruikt om als voorbeeld te dienen, BRIDE en 20-sim. BRIDE is gebruikt om de architectuur van het systeem te modelleren, en 20-sim is gebruikt om de algoritmes te modelleren. De robuustheid die nodig is voor autonome activiteit op lange termijn gaat verder dan de kwaliteit van software. Dit vereist algoritmes die om kunnen gaan met externe fouten.

Tolerantie voor externe fouten die afkomstig zijn van de verbinding van het cyberdomein en het fysieke domein, zoals een kapotte sensor, kan gecreëerd worden met een fout-tolerante regelaar. Falende hardware is onvermijdelijk in lange-termijn autonome activiteit; om deze reden is de mogelijkheid om zulke fouten te detecteren en er op te reageren essentieel voor een autonome robot. Fout-tolerante regelsoftware zoals hier gepresenteerd maakt het voor een mobiele manipulator mogelijk om fouten in een sensor, een motor in een joint of de aandrijving van een wiel op te vangen en zo zijn taak uit te voeren met eventueel verminderde prestatie.

De coördinatie van samenwerking tussen verschillende robots over een netwerk brengt een ander type externe fouten met zich mee, dat voortkomt uit de communicatie tussen software componenten. Terwijl netwerkcommunicatie het scala van mogelijke toepassingen van robots verbreedt, geeft het ook een aanzienlijke uitdaging om de interactie met coördinatie van samenwerking over een netwerk te regelen. Dit vanwege imperfecties in de communicatie. Een architectureel ontwerppatroon voor regelsoftware zoals hier voorgesteld, maakt het mogelijk om met imperfecties in de communicatie om te gaan, zodat het mogelijk is robots te bouwen die gedistribueerde faciliteiten nodig hebben en interactie hebben in een ongestructureerde omgevingen. De passiviteitslaag die hier wordt voorgesteld, vertegenwoordigt een nieuw soort veiligheidselement en kan als zodanig met iedere regelaanpak worden gecombineerd. Een toepassing toont de verbetering in robuustheid bij regelen met tijdvertragingen.

Regelsoftware van hoge kwaliteit, aangevuld met fout-tolerante algoritmes, is essentieel voor het bouwen van autonome robots die ontworpen zijn voor toepassingen in een open omgeving. Het bereiken van fout-tolerante regelsoftware van hoge kwaliteit is het doel van dit onderzoek en de combinatie van de gepresenteerde ontwikkelprocessen en fout-tolerante algoritmes streven dit doel na.

Contents

1

Introduction

1.1 Context

In the past decade, the use of robots has been increasing with an amazing speed. More and more applications are being proposed where robots are mounted on mobile platforms, not confined to designated safety zones and would work next to humans (Kranenburg-de Lange, 2012). This trend leads to increasing demands to safety and autonomy of a robot, which, in most cases, has to be resolved using its control software.

The quest for safety and autonomy of a robot is extremely complex, and strongly connected to concepts of reliability. Robots are designed to perform a variety of tasks in diversified conditions. This, on one hand, creates a great number of failure points, and on the other, numerous opportunities to complete the task in an alternative way, *e.g.* recover from a failure. The task to autonomously exploit these opportunities is designated to robot control software, which needs to contain algorithms to recognize faults and to respond to it appropriately. Moreover, the software itself has to be designed and produced in a way that it does not become a point of failure.

In modern robotic software design, there is a trend towards Component-Based Software Development (CBSD). CBSD promotes separation of concerns, i.e. encapsulation of particular functionality in a software entity called component. This practice assists re-use of "large-grained" pieces of software (Brugali and Scandurra, 2009; Brugali and Shakhimardanov, 2010). While CBSD allows to improve the scalability of a software system and reduce complexity of a single component, it increases demands on reliability and robustness of a component.

The work reported in this thesis has been performed in scope of the research project BRICS. This project was directed on summarizing and combining techniques and methodologies to increase quality of produced software in the robotics domain and speed of its production. One of the challenges raised on the way of creating component-based software is its reliability.

Subjects of reliability, safety and autonomy of a robot are not only complex due to complexity of robotic applications, but also due to intersection of different engineer-

ing domains often leading to confusion in used terms. Thus, before the concept of the thesis can be presented, it is necessary to clarify the terms and concepts that are being used in this work (Section 1.2). Furthermore, to clarify the motivation of the topics covered in this work the elements of fault taxonomy are analyzed (Section 1.3). The goals the thesis are described in Section 1.4, while the contributions are listed in Section 1.5. This chapter is concluded with the thesis overview (Section 1.6), which summarizes the research questions studied in the individual chapters.

1.2 Concepts of dependability

Terminology of dependable computing, which was most extensively presented by Aviezienis et al. (2004) is used in this thesis. In this section, some of the terms are recapitulated and the correlation of those terms with terms used in other domains related to robotics is given.

Understanding of the correlation between *fault, failures and errors* is essential for design of reliable systems. *Failure* is the term for an event of inability of the system to provide its desired service. The concept of a failure is binary in dependable computing, while in robotics, a notation of *quality of service* is often used as a term for percentage of service that has been fulfilled; and a *failure* is in that case indicates inability to provide a minimum *acceptable service* (Lussier et al., 2005). A failure is a result of *error* propagation. An *error* is a state of the system that may lead to a failure. A *fault* is a vulnerability of the system; it is a cause of the error. An event with leads to an error state due to a fault is termed an *activation of a fault.* In process engineering and some times in robotics, an event of fault activation is called an *abnormal event* as it is not part of nominal behavior. The correlation between terms is shown in Figure 1.1.

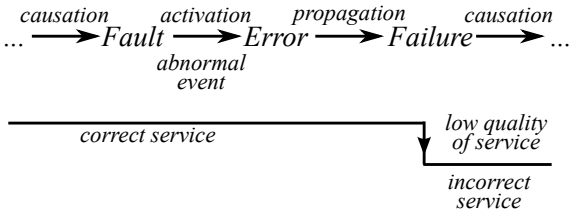


Figure 1.1: Fault propagation process based on Aviezienis et al. (2004)

Reducing the number of faults increases the reliability of a system and allows to maintain higher level quality of service. The means to achieve this reduction are termed *fault avoidance*. *Fault avoidance* methods are targeted on reducing the number of ways in which the system can be put into an *error* state. Fault avoidance includes means of *fault removal* and *fault prevention* (Aviezienis et al., 2004). *Fault removal* is a process of testing, analysis and modification of the system to remove system vulnerabilities. *Fault prevention* are means to preempt introduction of faults into the system which is normally achieved through the application of development methodologies.

Fault acceptance is a term for admitting the existence of faults in the system and find-

ing an alternative way to maintain the desired system reliability. *Fault acceptance* methods attempt to interrupt the fault propagation process (Figure 1.1) to increase the reliability of the system. Algorithms which can interrupt the fault propagation process are termed *fault tolerant*.

Fault forecasting techniques are aimed to justify the trust in the system's ability to deliver a service. If the risk of failure is considered to be unacceptable, the design procedure should be repeated until the system satisfies the requirements. Redesign of the system can be tackled by either fault removal or fault tolerance. Fault removal is the most direct contribution to dependability of the system, but in a robotic system it is not always possible due to an open environment. Fault tolerance contributes to dependability indirectly and some components are allowed to fail, but the system overall will deliver a correct service.

The concept of *robust autonomy* is an extension of fault tolerance; it allows some functionality to fail, but the system will retain other functionalities and will deliver an acceptable service.

1.3 Fault taxonomies

The improvement of system reliability starts form identification of points at which failures can be introduced. A developer has to identify faults that lead to the most catastrophic consequences, which are determined based on the damages involved and expected frequency of the failure. A developer would perform one of the fault forecasting techniques to identify the possible improvement points, removing dependability threats one by one from the system.

Similarly, to the developer of a system, we seek the dependability threats to a robotic system but in more generic sense. The fault forecasting techniques are based on the a priori/experimental knowledge available in the domain and are highly dependent on experience of engineers. To improve the process of failure analysis, the domain knowledge should be captured in the same manner. One of the best ways to represent knowledge is through hierarchical structures such as taxonomies. A taxonomy allows to identify the failures that are common to the majority of robotic systems, thus focusing the research on those that would give better contribution.

Two types of clustering faults form the taxonomy of Aviezienis et al. (2004) have to be recapitulated for future discussion.

First, faults can be classified as internal or external with respect to the boundary of a software component. Internal faults originate from the component itself, for example, from incorrect memory management. Majority of such faults can be removed from the component as a result of a thorough development process and application of supporting tools. External faults are violations of execution constraints, such invalid inputs, late message arrival or an unexpected behavior of peer components. External faults can not be removed during development of a component; however, additional algorithms can be added to the component to tolerate those, *i.e.* fault tolerance.

Second, faults can be classified as development or operational. The development

faults are always internal. The development faults in the software can always be treated with means of fault avoidance. The majority of operational faults are external as such can not be avoided and thus has to be accepted. The operational faults in the software have to be treated with fault tolerance means. Fault tolerance with respect to external operational faults is termed *robustness* by Aviezienis et al. (2004).

The taxonomy presented in here is orthogonal to the taxonomy of faults presented by Aviezienis et al. (2004). It is based on sources of faults in robotic systems. Information about the sources of the faults can be used to improve fault forecasting process and identify directions of research on improving robustness of a robot.

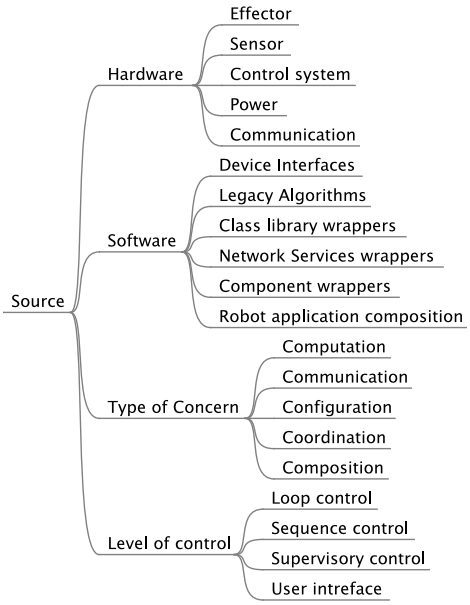


Figure 1.2: Taxonomy of fault sources

The fault analysis and classification are influenced by the chosen architectures, layer structuring and functional decomposition, thus not all elements of the taxonomy will be relevant to all types of robots. The sources of failure can be analysed from four different perspectives: hardware, software organisation, separation of concerns and control organisation, as depicted in Figure 1.2.

From a hardware point of view, there are 5 functional types of elements in a robot (Carlson et al., 2004):

- **Effectors** are elements representing parts with which robots can interact with the environment. All types of limbs, wheels or actuation mechanisms compose this category. Effectors include mechanical structures, actuation motors and wiring up to a controller.

- **Sensors** are elements responsible for perception. The wiring of the sensor to a controller is also included in this category; hence there is no difference from the control system point view between failure in the sensor itself or wiring to it.
- **Control systems** are elements responsible for computation of any kind. This type of element covers microcontrollers, computers and hardware that support the communication between parts of the control system on the robot.
- **Power** denotes any source of energy that is available for the robot as well as all power related wiring.
- **Communication** - Any devices that provide communication between the robot and environment. Examples would be tethers and wireless access points.

Another view on the source of failure is from software organisation. The software organisation for Generic BRICS Robot Application Software Architecture was proposed by Kraetzschmar et al. (2010). This architecture is based on 6 layers of abstraction. The failures accordingly can originate form each level of abstraction as presented in Figure 1.2.

Separation of concerns is a central concept in component based software development, as will be discussed in more detail in Chapter 2. It determines that types of functionality should be independent of each other. A way of clustering the functionality of a software system has been presented by Bruyninckx et al. (2013). Based on this clustering, the fault sources can be classified as follows:

- **Computation** stands for the algorithm implemented in the software. The corresponding faults are vulnerabilities that prevent the component from performing its main function to execute its main algorithm.
- **Communication** stands for information exchange between components of the software. The corresponding faults are vulnerabilities related to message passing between components, such as timely delivery of messages.
- **Configuration** stands for parametrization of the software. The corresponding faults are incorrect sets of parameters being provided to the component, which would prevent component from functioning properly.
- **Coordination** stands for states in which component can be, *e.g.* the discrete event logic of the component. The corresponding faults are vulnerabilities that would result in inconsistent states between cooperating components, essentially jeopardizing coordinated cooperation.
- **Composition** stands for structure of the software, *e.g.* the interconnection of the components. The corresponding faults stand for incorrectly determined connections between components.

Layers of control (Figure 1.3) offer a way to classify the fault sources and effects of the corresponding failures. Moreover, clustering fault sources based on the control layer-

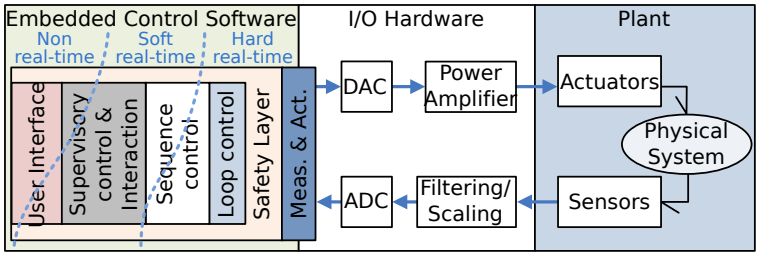


Figure 1.3: Layers of control (Broenink and Ni, 2012)

ing allows to identify possible ways to tolerate them. Each of the layers can contain or be a source of failures.

- **Loop Control** - This level is concerned with the dynamics of the system. Failures on this level result in the extensive forces and speeds, which could lead to destruction of the hardware or environment. Short response time is essential for fault tolerance on this level; recovery procedures can be reduced to suspending the execution.
- **Sequence Control** - This level deals with the trajectory execution by issuing commands to loop control level based on known trajectory. Failures on this level result in incorrect movements, which are similar to a slip for a human solving similar task (Carlson et al., 2004). Fault tolerance on this level is mostly a trade-off between speed and robustness of fault detection; the complexity of possible recovery procedures is limited by the speed of response.
- **Supervisory Control** - This level determines the long term strategy, it deals with a task execution planning by determining the sequence of actions. Failures on this level result in incorrect actions. Failures on this level are similar to human mistakes (Carlson et al., 2004). Fault tolerance on this level is concentrated on robustness of fault detection, correct probability estimates and high isolation properties; the main recovery procedures operate on this level.
- **User Interface** - This level is related to robot human interaction. Failures on this level result in incorrect directions to the robot. It is similar to a communication failure. Fault tolerance on this level is implemented through command confirmation and assistance requests.

Failures can be classified based on the type of deviation from normal execution flow. A taxonomy of failure types for the robotics domain are presented in Figure 1.4 :

- **Wrong execution path** - the failure of the component results in initiating an incorrect workflow, such as executing an incorrect part of the task plan.
- **Wrong timing** - the failure of the component results in delayed or a too early output. The examples could be congestion or change in the system dynamics.

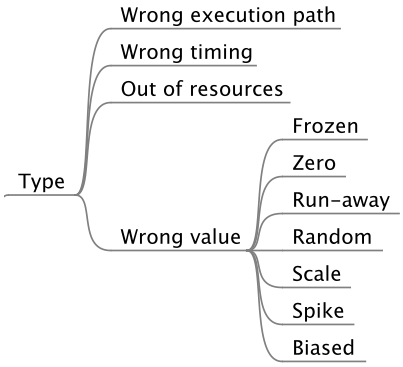


Figure 1.4: Types of failure (Sözer, 2009)

- **Out of resources** - the failure of the component results in the request of resources that do not exist or is unusual for this component. Examples are attempts to overload a CPU, memory failures or loss of power in batteries.
- **Wrong value** - the failure of the component results in wrong output value. Examples are sensors that give incorrect output, joints that do not respond to control system commands or incorrect computation results.

1.4 The goal

The goal of this work is to study ways to improve reliability and robustness of the robotic applications through improving dependability of software. The most specific part of robot's software is its motion control as it determines the way a robot affects physical world during completion it task. This is one of the vital parts of a robot, where non-functional requirements such as robustness and reliability are of most importance. A robot's motion control ensures the ability to proceed with designated task.

The goal of this work can be divided into sub goals:

- Devise a methodology to reduce number of internal faults in the robot motion-control software, that would compliment existing development processes.
- Study ways to reduce effects of external faults, fault that:
  - originated from robot hardware, *e.g.* sensors, motors;
  - originated from connection between software components.

1.5 The contributions

The major contribution of this work can be summarized as:

- A methodology for software development that uses "uniform" modeling of the software components to improve reliability. The presented methodology ex-

ploits the software practice, separation of concerns, to advocate modeling of software form different perspectives as means to produce reliable software components. The proposed methodology is supported by the developed tool chain.

- An approach for increasing the resilience of a mobile manipulator to sensor failures. Fault-tolerant control strategies are presented that allow a mobile manipulator to continue operation despite failures of a sensor or motor in a joint or a wheel drive. The example demonstrates that faults can be efficiently compensated in parallel kinematic structures and open kinematic chains.

- An architectural pattern for control software that allows to tolerate communication failures. A new control paradigm emerged from tele-operation research (Franken, 2011) was used to construct an architectural pattern for control software that can tolerate communication failures such as delays or loss of communication. An application example demonstrates the gain in robustness with respect to time delays.

1.6 The overview

The thesis contains three chapters each dedicated to the chosen goal. Each of the chapters is styled as a standalone work, which can be used independently of the other chapters. The related literature is reported within each chapter separately. The intermediate results leading to work presented in the chapters of this thesis were published as public EU project BRICS deliverables D6.1 and D6.2 (Brodskiy et al., 2011, 2013). Appendix A was published in proceedings Workshop of SIMPAR2010 (Dresscher et al., 2010).

Methods of fault avoidance in development process are discussed in Chapter 2. The position of model-driven engineering in component based software development is discussed. The concept of modeling the Computation and the way to gain software reliability improvement is presented. The development process is discussed. The combination of existing tools into a tool-chain is presented, and resulting tool-chain is demonstrated by a use case.

The focus of the Chapter 3 is tolerance of the hardware faults of robots such as sensors and motors. The elements of fault tolerant control are discussed. An approach to fault detections and isolation combined with a recovery method is presented. The proposed architecture is implemented for the mobile manipulator, KUKA youBot. The simulation results are presented demonstrating the gain in reliability with respect to failures in the robot's hardware.

The focus of the Chapter 4 is tolerance of communication faults. The described method is based on the idea of energy-based constraints and the so-called passivity layer, which are used to define an architecture of a single controller component and composition controller-components. A use case is used to demonstrate the effect of communication imperfections on controller performance and a improvement in performance achieved by including a passivity layer into the controller software of the experimental setup.

The conclusions of the thesis, the outlook into future research topics and application of the results are presented in Chapter 5.

2

Fault Avoidance in Development of Robot Motion-Control Software by Modeling the Computation

In this chapter, we discuss the process of modeling the Computation as means to increase reliability of software components. The approach to developing Embedded Control Software (ECS) is tailored to Component-Based Software Development (CBSD). Such tailoring allows to re-use the ECS development process tools in a development process for robotics software. Model-to-text transformation of the ECS design tool is extended to model-to-component transformation suitable for CBSD frameworks. The development process and tools are demonstrated by a use case.

2.1 Introduction

Improving quality of a product is a major goal of a development process. A development process is typically tailored to a chosen type of product to efficiently achieve this goal. Robotic application development combines results of many domains: AI, software, mechanical, electrical and control engineering. The corresponding development process has to ensure an efficient combination of these domains in a single product. Such development process combines practices of development processes from these domains.

In here, we present a combination of two relevant development processes: The Robotic Application development Process (RAP) (Kraetzschmar et al., 2010) and the Embedded Control Software (ECS) design trajectory (Broenink et al., 2007, 2010). The proposed combination is motivated by a desire to gain re-usability and reliability of robot control software as suggested in the corresponding development processes. The proposed combination also allows to integrate the tools that support these development processes.

The Robotic Application development Process (RAP) has been proposed by the BRICS project as a result of simultaneous tailoring of over 25 different development processes to the robotics domain and CBSD. The focal point of this development process is structuring of software to maximize its re-usability and flexibility, using CBSD techniques.

The ECS design trajectory represents practices in development of software for embedded applications. It advocates modeling of the algorithm of a software component (controller) with a consequent automated implementation process. The focal point of this development process is the reliability of the developed software.

The chosen development processes use modeling of software to achieve a higher quality product. Nevertheless, these development processes have different perspectives on software modeling, thus resulting in different types of models being used. We argue that combining these models result in a uniform description of the software on the modeling level thus allowing to gain the benefits from both development processes.

Throughout this chapter, the combination of these two development processes is discussed. Relevant terms used in model driven engineering in application to component based software are introduced in Section 2.2. The motivation for modeling the Computation to gain software reliability improvement is presented in Section 2.3. The combination of RAP and ECS design trajectory is discussed in Section 2.4. The integration of tools used in RAP and the ECS design trajectory is presented in Section 2.5. The resulting tool-chain is demonstrated by a use case in Section 2.6.

2.2 Model-Driven Engineering in Component-Based Software Development

Current research in software development for robots focuses on Component-Based Software Development (CBSD) (Brugali and Scandurra, 2009; Brugali and Shakhimardanov, 2010). CBSD supports the development and reuse of large-grained pieces of robotics software through component-based software frameworks such as ROS (Willow Garage, 2013), Orocos (Bruyninckx, 2001; Orocos Project, 2013), SmartSoft (Schlegel, 2013; Schlegel and Worz, 1999). To gain the advantages provided by a component-based software framework, the software has to be structured into independent components. The structuring of software into independent components requires structuring on the system level, *i.e.* a composition of components and the component level, *i.e.* an internal component structure. This need for structuring of software has triggered research on the application of Model-Driven Engineering (MDE) techniques to CBSD.

A model is an abstract representation that provides a simplified description of the system omitting information irrelevant to the problem, thus allowing to focus on chosen properties of the system. Depending on the purpose of the model, the same system is represented by different models. For example, various professionals have their own distinct design goals during the design of a car (Figure 2.1), thus, aspects they are interested in differ, and the models they use are different. Similarly to a car, a software component can be modelled from various perspectives depending on the purpose of the model.

The information captured by a model is restricted by the chosen set of modeling primitives. The set of modeling primitives is the language of the model; it is also referred to as a domain-specific language (DSL). These modeling primitives and relations between them are a meta-model (a model of the model), which represents the next level

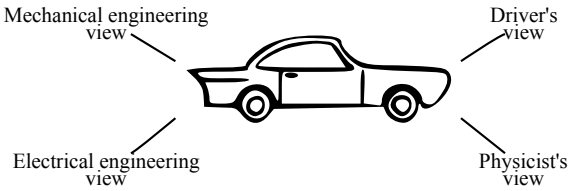


Figure 2.1: Various perspectives (models) of the same object

of model abstraction. The Object Management Group (OMG, 2013) has proposed four levels of model abstraction, defined as (Figure 2.2):

- **M3 – Meta-meta-model:** The highest level abstraction, which provides constraints and restrictions a meta-model has to satisfy without using any domain-specific features.
- **M2 – Meta-model:** Domain-specific language, which provides modeling primitives (e.g. elements, rules & constraints between them) from the chosen engineering domain.
- **M1 – System model:** A model of the (robotic) system, but without using a specific implementation details.
- **M0 – Real system:** Implementation of the model.

The levels of abstraction have a "conform to" relation. A "conform to" relation indicates that models of a lower M-level use elements and composition rules defined by the model of the higher M-level. Since a model focuses on representing a single aspect of the system, the system can conform to multiple models on higher levels of abstraction (Figure 2.2).

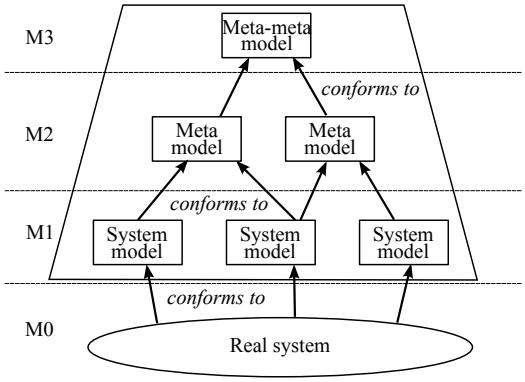


Figure 2.2: Four levels of model abstraction by OMG (2013)

As presented by Bruyninckx et al. (2013), the modeling primitives of the component model can be grouped in the 5C categories:

- **Computation** stands for the actual algorithm implemented in the component. Typical examples of computation models include logic circuits, transfer functions etc. An example of Computation is a control law such as a PID controller.
- **Communication** stands for information exchange between components. The description of component interface and communication constraints (exchange time, bandwidth, latency, accuracy, priority, etc.) is type of information covered by the communication model.
- **Coordination** stands for states in which component can be, *e.g.* the discrete event logic of the component. The coordination model is typically expressed using Finite State Machines (FSM).
- **Configuration** stands for parameters of the algorithm implemented by the component. An example of Configuration is parameters of PID controller.
- **Composition** stands for interconnection of the components. It contains information on dependencies between components and organization structure.

MDE employs models in several ways: design and analysis of the software system, and automated creation of software artifacts. Models, due to their focus on specific aspects of the system, offer a better overview and understanding of the system. A model, provided that it has model checking or simulation capabilities, can be used for confirmation or proofing properties of the system, such as the stability of a control system or absence of deadlocks in software. Computer-Aided Design (CAD) tools, used in support of the modeling process, are also used to automate the implementation process, *i.e.* the transformation of the model into the executable program code.

In CBSD, models are used to build the software architecture (Schmidt, 2006). The goal of building a software architecture is reflected in type of models that are used and supported in CBSD. Software architecture is described by models formulated in terms of Composition, Connection (Hochgeschwender et al., 2013), Configuration (Brugali and Scandurra, 2009) and Coordination (Klotzbucher et al., 2013). Modeling the Computation is not part of the existing CBSD practice. As motivated by Kraetzschmar et al. (2010), the Computation is not modeled explicitly in order to preserve the flexibility that is provided by general-purpose implementation languages.

Although general-purpose implementation languages are preferable for development of a complete application in non-component based way, it is not the case for development of software components, that do not require access to the specific features of the operating system or hardware. An appropriate functional decomposition separates components that exploit such features from those that do not. The software components that focus on the algorithms required for the application and do not require access to specific features of operating system or hardware directly can be more efficiently described using DSL. In other words, modeling the Computation of such components would result in software quality gain without loss of flexibility.

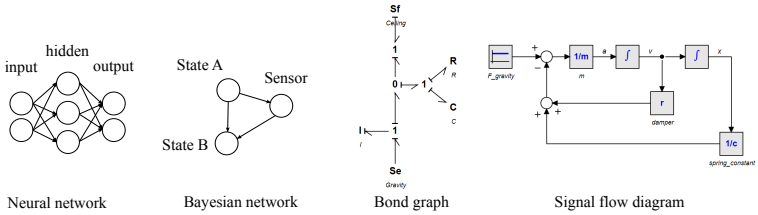


Figure 2.3: Various computation models

2.3 Modeling the Computation to increase software dependability

ECS engineering practices rely on modeling the Computation to ensure high reliability of the produced software. Modeling the Computation is part of a development process, in which a model of the algorithm is made. The Computation model represents the mathematical nature of the algorithm. A model leaves out a platform, an operating system, a framework or programming-language specific elements. A computation model can be constructed based on several different meta-models: transfer functions, logic circuits, Bayesian networks, neural networks, bond graphs etc (Figure 2.3).

Modeling the Computation as described in the ECS trajectory increases reliability of the developed software (Broenink et al., 2010). A model of Computation, or the model of the algorithm, serves three purposes:

- A model improves understanding of algorithm functioning, thus revealing the points where robustness should be improved.
- A model, supported by simulation tools, can be used to study a behavior of the algorithm, in an attempt to verify its qualities.
- A model can be automatically transformed to a formulation needed for the next step in the development process, for example, into a usable software artifact (e.g. as a component), if model is sufficiently detailed and automatic transformation is supported by the employed tools.

These purposes of a computation model overlap with the means of improving a system's reliability, which are typical for a development process: fault prevention, fault removal and fault forecasting (Figure 2.4).

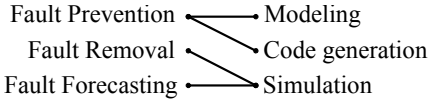


Figure 2.4: Mapping of reliability means on purposes of a model

Modeling the Computation provides fault prevention on two levels, namely design and implementation.

In the design phase, a developer models the Computation to obtain a simplified but competent representation of the algorithm (the model of algorithm). This is achieved by choosing the modeling language that supports an effective and simple representation of the algorithm that is being developed. In other words, the chosen modeling language is more expressive than a general-purpose implementation language. For example, a Bayesian network diagram (Figure 2.3) can be used to build decision logic in the application, and it represents many of lines of code required for its computation. Thus, the general-purpose language might, in this case, obscure the design intent, while a model allows to focus on design of the algorithm avoiding implementation issues (Schmidt, 2006). The model of algorithm provides a clear expression of the design intent and thus prevents logic faults. It especially concerns the class of development, human-made, deliberate, nonmalicious faults (Aviezienis et al., 2004) as the trade-offs made at development time are made explicit.

Appropriate tool support during modeling the Computation also allows to prevent development, human-made, non deliberate, nonmalicious faults in the implementation phase. The composition rules, defined by the meta-model, can be used to prevent illegal constructs that would result in faults, which are typical unintended actions, done by mistake. The enforcement of composition rules on model level also prevents faults occurring due to misunderstanding between collaborating teams (Broenink et al., 2012), which also classify as non-deliberate, human-made faults. Furthermore, an appropriate tool support allows to automate the transformation of the model to a formulation needed for the next step in the development process. This allows to reduce significantly the number of human-made faults as the developer is excluded form the transformation process; thus it prevents the introduction of faults into software at this step of the development process.

Reduction of the number and severity of faults is achieved in the development process through a sequence of steps: verification, diagnosis and correction. This fault removal process is similar to the approach of iteratively refining models, used in the ECS design trajectory (Broenink et al., 2007, 2010). Examination of the system behavior is necessary for the verification process; thus a simulation capability by the tools supporting the modeling process is needed. Simulation allows to obtain system behavior without completing the implementation step, and allows to preform a detailed inspection of the generated behavior. In simulation, a designer has full control over the system states, outputs and inputs, thus any situation or behaviour can be tested, and the response can be verified. An example of such use of simulations is fault injection or fault modeling (Broenink et al., 2012). This approach prescribes building a model of a faulty behavior in addition to a model of expected behavior. The designed algorithm is then tested on the model with a fault to assess the response of the control algorithm. This approach is used in Chapter 3 to evaluate performance of the proposed fault tolerant control. Overall, use of simulateable models simplifies and increases the quality of the fault removal process.

The possibility to simulate and inspect behaviors of an algorithm also contributes to fault forecasting. Fault forecasting aims at estimating the present number, the future incidence and the likely consequences of faults. A qualitative evaluation is performed

to identify the combination of events leading to possible system failure and predict, classify and rank failure modes. Quantitative evaluation determines, in terms of probabilities, the extent to which some of the attributes (i.e. measures) are satisfied. Similarly to the fault removal process, an ability to simulate the model allows to examine a wider state space of the system, thus studying quantitative effects of faults. For example, dynamic models of the robot hardware (plant) can be used to check the consequences of failing hardware components for the algorithm.

2.4 Modeling the Computation in the Development Process

Modeling of software in a development process can be done from different perspectives (5C Section 2.2), moreover modeling software from each perspective leads to improvement in software quality. Thus by modeling software from all perspectives allows to maximize the benefits from using models in development process. A development process has to indicate at which step each model has to be developed. Furthermore, the steps at which the models are being synchronized and transformed into implementation have to be indicated such that inconsistencies in models are avoided.

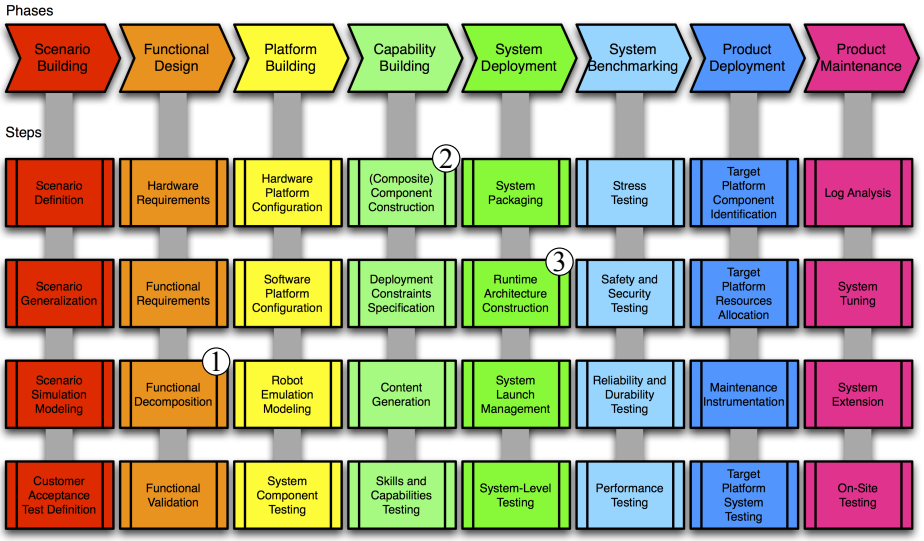


Figure 2.5: BRICS Robotic Application development Process (RAP) (Kraetzschmar et al., 2010)

An overview of RAP is given in Figure 2.5. RAP consists of 8 phases with 4 steps per phase. A modeling process embedded into RAP mostly focuses on modeling perspectives related to software system design (Kraetzschmar et al., 2010). Composition and Configuration models are developed in Functional Decomposition step (nr. 1 in Figure 2.5) and implemented in Composite Component Construction step (nr. 2 in Figure 2.5). Communication model is developed in Run-time Architecture Construction step (nr. 3 in Figure 2.5). Modeling the Computation is not covered in RAP.

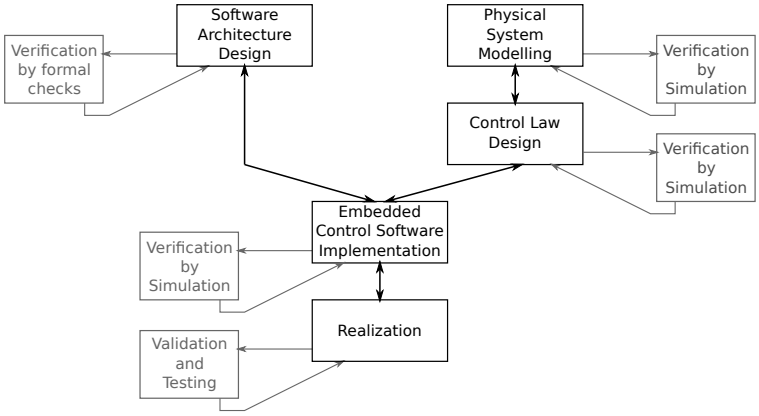


Figure 2.6: ECS design trajectory (Broenink et al., 2010)

To incorporate modeling the Computation into RAP, it has to be combined with another development process, such as the ECS design trajectory. The ECS design trajectory advocates an iterative refinement of a computation model of a controller from a basic functional and conceptual model to the concrete algorithm that can be automatically generated into the implementation, i.e. executable code. It also prescribes the use of a dynamic model of the system to perform verification by simulation at every step of the development process (van Amerongen and Breedveld, 2003; Broenink et al., 2007), while the model of Computation is (re)formulated until a desired behavior is achieved.

The scope of the ECS design trajectory is comparable to the *Platform Building*, *Capability Building* and part of the *System deployment* phases of RAP. The *Capability Building* phase corresponds to design and implementation of the component algorithms, which include creating of an implicit model of Computation. The integration of the ECS design trajectory into RAP enables, due to its emphasis on modeling of Computation, to model explicitly a component from all perspectives (5C, discussed in 2.2). Tailoring of the methods and integration of the tools will increase the reliability of the resulting software, as was motivated in the previous section. How the steps of the ECS design trajectory map on steps of RAP is presented in Figure 2.7. The step, where modeling the Computation performed, is emphasized.

The ECS trajectory combined with RAP is present in Figure 2.8. Note that the updated ECS trajectory is only centered around the the steps indicated in Figure 2.7, preceding and consecutive steps have to be performed as it is prescribed in the original RAP model. The updated ECS trajectory process combines the modeling procedures of the ECS trajectory and RAP, thus providing a uniform modeling of a software component that covers all 5 aspects of a software system (5C Section 2.2).

The updated ECS trajectory combined with RAP indicates two additional interlinks

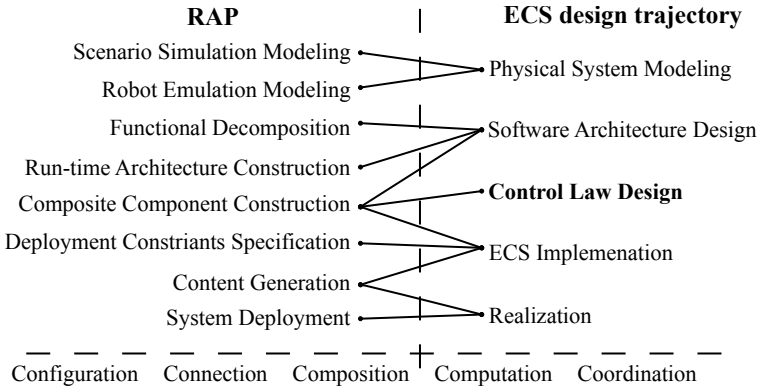


Figure 2.7: Correspondence between the ECS design trajectory steps and RAP steps. The modeling the Computation is done in the emphasised step.

between architecture design and algorithm design. These interlinks allow to improve designed algorithms and architectures.

The line nr. 1 (Figure 2.8) indicates that the constrains imposed by the runtime architecture has to be taken into account during Algorithm design. The analysis of the effects of the run-time architecture on the algorithm performance allows to forecast possible failures and develop algorithms that tolerate those. For example, the runtime architecture determines timing effects of inter-component communication, which

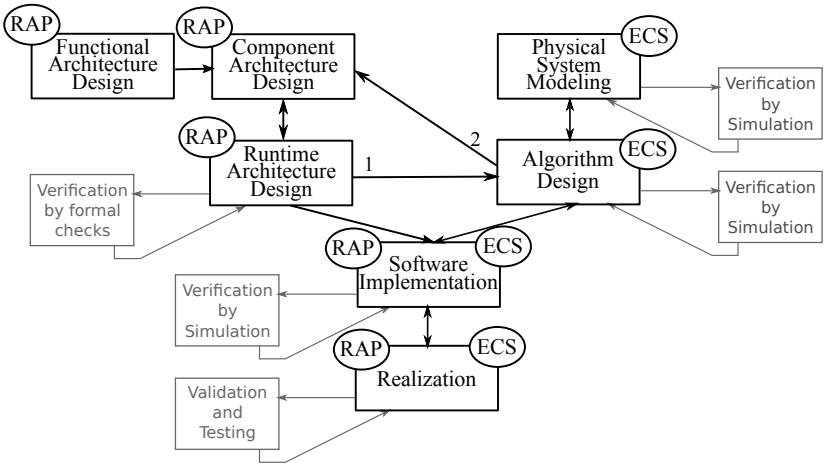


Figure 2.8: ECS trajectory combined with architectural elements form RAP, an update of Figure 2.6

may change in algorithm performance. Inclusion of these effects allows to identify such issues and correct either architecture or algorithm.

The line nr. 2 (Figure 2.8) shows the information flow from the *Algorithm Design* to *Component Architecture Design*. The distribution of the algorithm elements into components is determined by the component architecture. However, a detailed knowledge about algorithm functioning might affect the component architecture, as the initial component architecture might impair algorithm performance. For example, the original component architecture might result in unreasonable communication requirements (such extreme band-with or un-achievable small latency). Inclusion of component architecture into Algorithm Design step allows to identify these issues at modeling stage and can lead to restructuring the component architecture.

At the *Software Implementation* step, the models of the algorithm (Computation or Coordination aspects developed in the *Algorithm Design* step) are combined with the model of software architecture (Configuration, Connection, Composition aspects developed in the *Runtime Architecture Design* step). The automation of the *Software Implementation* step is discussed in Section 2.5.

Both the development processes encourage the use of modeling tools. However, due to their emphasis the suggested modeling perspectives are different (Figure 2.7 bottom line). Therefore, to support modeling of the software system from all perspectives, the modeling tools have to be integrated.

2.5 Tool integration

The updated design trajectory, shown in Figure 2.8, indicates the point of integration of the tools (*Software Implementation* step). The goal of the tool integration is automation of the *Software Implementation* step, which requires transformation of the algorithm model into forms/artifacts necessary in the consecutive steps of the development process.

The models of a component are obtained in the *Algorithm Design* step and *Software Architecture Design* step. These models are competent for the automated *Software Implementation* step if they contain sufficient information about all the 5 aspects of a software system (5C section 2.2). During the *Software Implementation* step the models of the algorithm and model of the Run-time architecture are combined. The *Realization* step, following the *Software Implementation*, is equivalent to *System Deployment* phase of RAP (Figure 2.7) and deals with transferring the implemented components on to real robots using the model of Run-time architecture.

The development process (Figure 2.8) can be used with a number of tools that provide similar functionality such as presented in Table 2.1. Tool integration requires passing of information between tools of left and right columns. Since tools noted in the right column are also used to support the *System Deployment* phase (Figure 2.5) of the development process (part of *Realization* step in Figure 2.8), it is sufficient to pass models from right to left. The information passed between tools depends on the type of integration.

Configuration & Connection & Composition	Computation & Coordination
BRIDE[1]	MatLAB[2]
SmartSoft[3]	**20-sim**[4]
OpenRTM[5]	LabView[6]
Proteus[7]	OpenModelica[8]
TERRA[9]	Dymola[10]

Table 2.1: Tools and their focus in modeling

To exemplify the tool integration, the tools BRIDE and 20-sim are used respectively as suggested by the authors of RAP (Kraetzschmar et al., 2010) and the ECS design trajectory (Broenink et al., 2007).

2.5.1 BRIDE as CAD for CBSD

The BRIDE toolchain (BRICS, 2013) was developed by the project BRICS to support RAP. This toolchain facilitates the design of (stand-alone) components and component compositions for robotic application deployment. The BRIDE tool offers graphical editors to model components using the BRICS Component Model (BCM) (Bruyninckx et al., 2013) or any of several software frameworks (Orocos RTT, ROS) (Orocos Project, 2013; Bruyninckx et al., 2003; Willow Garage, 2013). For the abstract BCM model, the BRIDE toolchain offers model-to-model transformations into the target frameworks (Figure 2.9 line M2M).

Furthermore, BRIDE supports modeling primitives in Communication, Configuration, Composition and (to a lesser extent) Coordination. However, for modeling the Computation (i.e. algorithms) it offers *no* modeling primitives. The Computation (and Coordination) is directly specified in the target framework's implementation language, which is typical for tools of this type (Table 2.1), thus a programming language like C or C++ is used.

For transforming models into implementations, BRIDE offers model-to-text transformations (Figure 2.9 line M2T), also known as code generation. In this process, the toolchain is used to generate the target framework code for the used primitives, plus empty functions for the Computation. These empty functions contain protected regions, such that changed models do not overwrite manually added code. A component developer has to fill in empty functions of the generated component(-hull).

[1] BRICS (2013)
[2] MathWorks (2013)
[3] Schlegel (2013)
[4] Controllab Products B.V. (2013)
[5] OpenRTM Project (2013)
[6] National Instruments (2013)
[7] Proteus Project (2013)
[8] Open Source Open Modelica Consortium (2013)
[9] Bezemer et al. (2011)
[10] Dassault Systemes AB (2013)

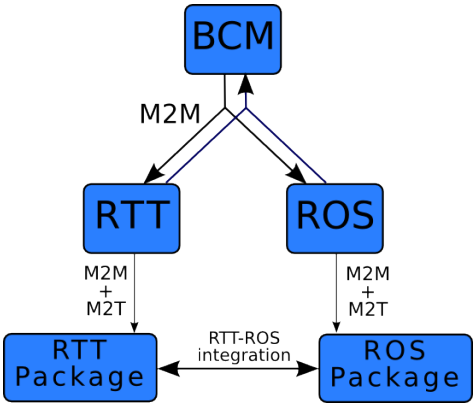


Figure 2.9: Overview meta-models and frameworks supported by BRIDE (BRICS, 2013)

Looking at the diversity in the available models of computation (e.g. state machines versus data-flow).

2.5.2 20-sim for Computation modeling

20-sim was developed for the design of mechatronic systems and supports the ECS design trajectory as development methodology. It embodies the concept of concurrent design of mechanical, electrical and control parts of the system. 20-sim provides modeling primitives for designing a component's Computation. This tool can be used for both modeling of control algorithms and physical systems behaviour. 20-sim allows a graphical approach to hierarchical structuring of an algorithm. It offers two ways of describing computation: declarative and imperative.

For automation of an implementation step, 20-sim offers model-to-text transformations. It employs a template-based code-generation to export a resulting algorithm into a high-level programming language (C, C++). A provided template specifies the form in which the algorithm will be represented. The template is an application with empty functions, which are marked up with special tokens. These tokens are replaced by the algorithms computation code during the code generation process Broenink et al. (2010).

2.5.3 CBSD- and Computation-model integration

In an overview of integration methods of model-based tools presented by Kapsammer et al. (2006), the authors suggest a model-(transformation-)based type of tool integration. It is presented as a low-maintenance cost and easily scale-able approach. In this method, on the basis of model transformations (relevant) data is imported or model-interfaces are determined.

Model-based tool integration is based on the concept of meta-models (Section 2.2), which are used to define the transformations between models generated by each tool.

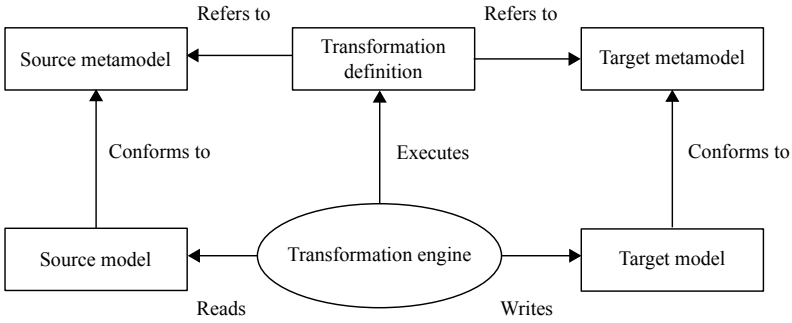


Figure 2.10: Basic concepts of model transformation (Czarnecki and Helsen, 2006)

The model-to-model transformation process is a translation of a source model into a target model (Figure 2.10). This transformation process is based on the source and target meta-models, whereby the meta-model primitives need to be translated. The union of two meta-models (sets of primitives that describe the same concepts) can be translated one-to-one. Such translation completely preserves the information in the target model and can be used for further modeling in the target tool. The difference between the sets of primitives of the source meta-model and the target meta-model is the information loss. This information loss is due to the fact that the target meta-model can not describe the same concepts as is possible in the source meta-model.

The transformation of any Computation meta-model into a component meta-model used by the CBSD tools (like BRIDE) results in a model reduction since the concepts of computation are not supported by such tools. To preserve the computation elements a second artifact is generated – the code (Figure 2.11). The combination of the generated code and the model completely represent a single component which can be used in the next phase of the development process, i.e. *System Deployment*.

The transformation of the Computation model into a software component can be automated using code generation (Figure 2.11). The model of the component has to be exported for tool integration. The approach taken here involves simultaneous generation of two artifacts: the executable code and the model of the component. The resulting component code conforms to the component model used for software-system building in the BRIDE and Computation model designed in 20-sim.

The integration between BRIDE and 20-sim is structured as a 2-step process (Brodskiy et al., 2013). First 20-sim transforms the Computation model into a code and a model of the component (Figure 2.11). Second, the model of the component is analysed and, by using predefined rules, a BRIDE model of the component is generated. The newly generated model contains information about the Communication and Configuration (component interface). The Coordination and Computation of the component are only encoded in the component implementation as BRIDE does not offer primitives to describe these concepts.

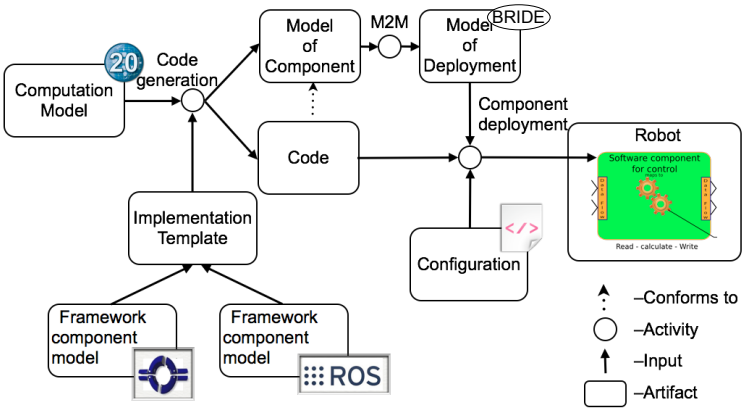


Figure 2.11: Model-to-component transformation

A template for code generation of a component (*Implementation template* in Figure 2.11) has to maximize openness and flexibility of the resulting code. As was presented in Section 2.2 there are 5 aspects of a software component (5C) that have to be exhibited to a developer to maximize possible re-use of the software component. Therefore, a template has been designed in such way that each aspect modeled in the source model is extracted and formed into a representation available in the chosen framework or target model. More specifically, in case of integration of the 20-sim and BRIDE, the model of the component is composed based on the template. This model contains the Configuration and elements of Connection (the component interface) while Computation and Coordination are encoded in code. The generated model of component is imported into BRIDE and then can be completed by including it into the Composition model of the application.

2.6 Use case application

2.6.1 BRICkS stacking application

The use case is centered around motion control for a mobile manipulator, the KUKA youBot (Bischoff et al., 2011; Brodskiy et al., 2011). The resulting set of software components is termed as *motion stack*. The software components are implemented using the methodology described above. Orocos RTT was used as target component-based software framework.

The robotic application that has been developed is autonomously stacking wooden bricks in a small box using a youBot (Figure 2.12). More specifically, only the bare youBot and no additional sensors (like a camera) are used. By monitoring the torques on the joints and the force on the tool-tip, the software can reason about the next task. Initially, the human teaches the robot where the position of the box is and thereafter feeds the bricks to the youBot sequentially. The youBot then places the bricks autonomously in the box. To achieve maximal filling of the box, the bricks should be

placed in an ordered manner. This application is chosen because it requires a variety of operating modes and complex interaction with the environment. A variety of operating modes require Coordination type of component in the system for the orchestration of the task execution. Complex interaction with the environment requires a Computation type of component for the control of the robot.

(a) The youBot **(b)** BRICkS **(c)** Box

Figure 2.12: Elements of the use case

The use case was intended for refining of methods for increasing component reliability especially with respect to robot control. Following the methodology as described (Figure 2.8), the use case has to accomplish the following steps:

- Functional Architecture Design;
- Component Architecture Design;
- Run-time Architecture Design;
- Physical System Modeling;
- Algorithm Design;
- Software Implementation;
- Realization.

Based on Figure 2.8 some of the steps can be performed in parallel. The Physical System Modeling step can be preformed in parallel to the Functional Architecture Design step. The Component Architecture Design, Run-time Architecture Design and Algorithm Design steps have circular dependency, and therefore require iterative approach. The Software Implementation and Realization steps are preformed sequentially.

The Physical System Modeling step is presented in detail in Appendix A, and a detailed description of the algorithms is provided in Appendix B. The focus of this section is the architectural design (Section 2.6.2), the effect of architecture on algorithm (Section 2.6.3) and automated transformation of the models into components (Section 2.6.4).

2.6.2 Architecture Design

The architecture design is done in three steps, each increase the level of detail at which system is described. First, the functional architecture is formulated to define the basis for distribution of functions between components. Second, the component architecture is devised to prescribe distribution of algorithm elements over components. Third, the run-time architecture is designed to define deployment constraints and communication details. The latter steps can be also used to improve the quality of the components' algorithms.

A generic functional-architecture of a motion control software (a motion stack) is presented in Figure 2.13 (Brodskiy et al., 2013). The proposed organization of the software specifies the granularity of the algorithms and provides a standard functional decomposition into components. This standardisation of component functions enables further harmonization of the component interfaces. Harmonized interfaces are a major requirement for introducing variation points or variability (Brugali et al., 2012). A variation point is a designated part of the architecture where various algorithms can be interchanged to alter the behaviour of the robot. For such point only the interfaces and operational semantics (such as update rates or data types) should match. As a result, an exchange of a component for one with a different behaviour should not require modifying the motion control structure nor any component internals.

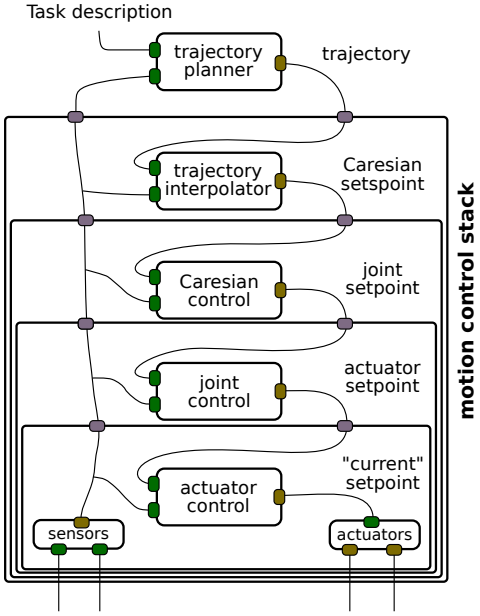


Figure 2.13: Conceptual architecture of the motion stack

Figure 2.14 depicts the component-level architecture of the motion stack, in the form of a data-flow diagram, based on functional decomposition presented above. The pre-

sented architecture is an example of a variability solution (Brugali et al., 2012). The "lowest" level of the control, i.e. the actuator control (Figure 2.14), is implemented in the firmware. The hardware drivers in the actuator control make the functionality provided by the firmware available to the rest of the software components. The joint control consists of two parts: independent joint position control and a kinematic map of forces on the tool tip to torques on the joints. This additional functional decomposition was created to fit the requirements of the use cases: switching between different types of joint control at run-time. The Cartesian control defines a potential field around the desired set-point in space, which then is transformed into a force on the tool tip. The trajectory planner and trajectory generator are shown as constant set-point generators in Figure 2.14 (see for instance *ArmJointSetpoint*) since these are task dependent components and can be varied.

The defined component architecture specifies most of the composition elements of the system. However, a run-time architecture is required to define how components are executed with their time and concurrency relations. A run-time architecture can introduce various effects that will affect the algorithm performance. For example, synchronization of the components may result in dead-lock or life-locks. Message passing between components can introduce time delays, or message losses. To verify the algorithm's robustness and tolerance of such effects, the robot model has to include them. Time delay elements (z^{-1}) indicate possible time delays in the signal exchange, that has been used in algorithm verification.

2.6.3 Algorithm Design

Algorithm Design (Modeling the Computation) goes in parallel with the design of the architecture. Architectural elements are used to structure the algorithm. The effects that are introduced by the architecture are used to model and identify unexpected behaviours. Knowledge about the algorithm performance is used to modify the component architecture and generate deployment constraints.

The algorithm design requires a competent model of the system that will be controlled. For motion control of the youBot, a model of the robot dynamics was developed (detailed description in Appendix A). This model was used to verify performance of the algorithms in a simulation. Animation facilities of the simulation program (20-Sim) and traces of the overall system states are used to study behaviour. Figure 2.15 shows a picture taken from the animation.

The data-flow diagram, shown in Figure 2.14, has resulted from the architecture design route. Each block of this diagram has to become a software component in the real system and has been filled in with required algorithm according to its task, details of the algorithms are reported in Appendix B. These algorithms are the Computation models of the future software components.

With all algorithms being modeled, the computation model of the motion stack, represented by the data-flow diagram Figure 2.14, can be simulated to study the behaviour of the robot. By performing simulations of this model, the algorithm performance and robustness can be assessed. For example, issues regarding message loss between com-

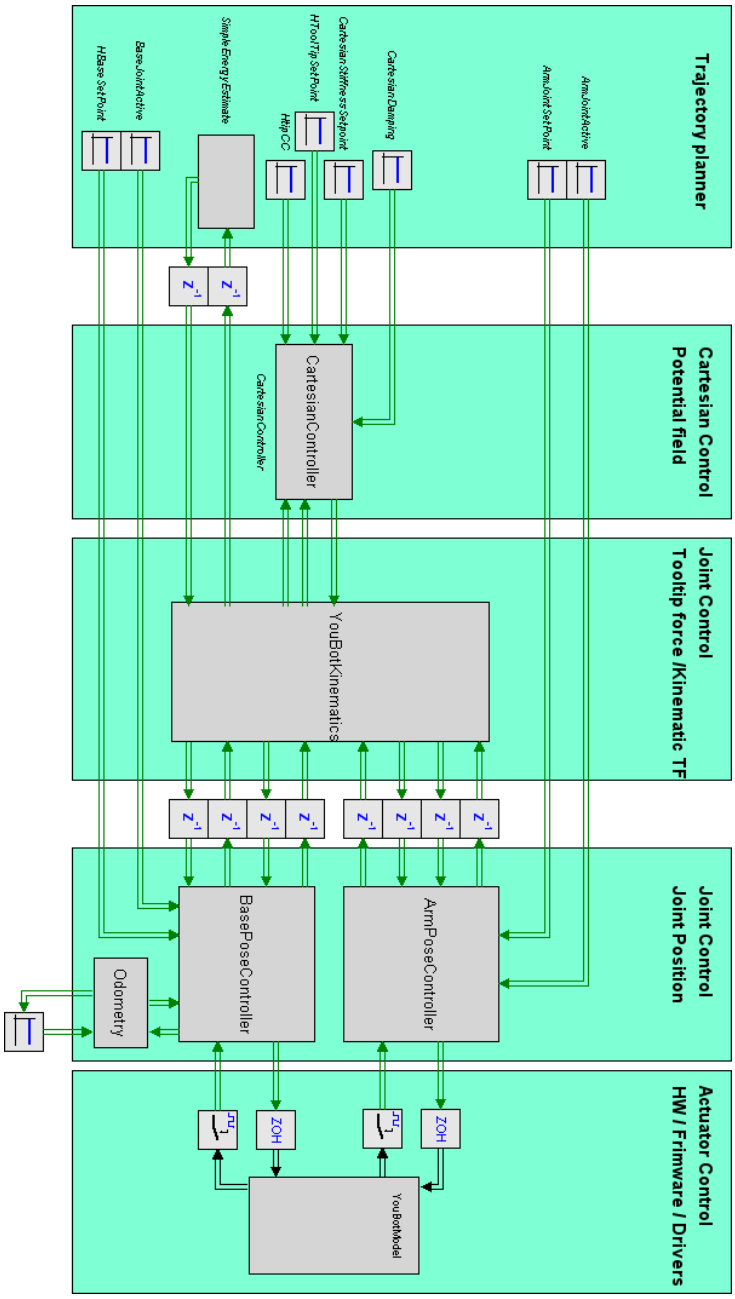


Figure 2.14: Architecture of implemented motion stack

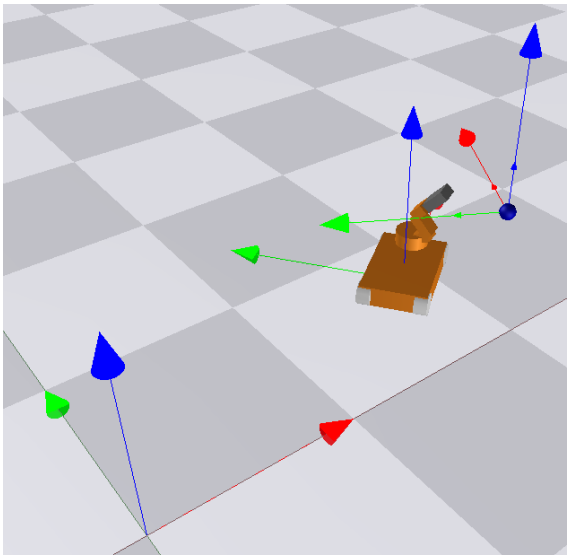


Figure 2.15: 20-Sim animation of KUKA youBot simulations

ponents can be studied or the simulation can be used for parametric optimization of the algorithm.

When the responses of the system meet the task requirements, such model is ready for the Software Implementation step. As noted in Section 2.5, the model of algorithm is competent for automated code generation when it includes architectural and computation elements.

2.6.4 Software implementation

Orocos RTT was chosen as the target component-based software framework for implementation of the motion stack as one of the frameworks supported by BRIDE. The parts of the motion stack were implemented as components for the chosen framework. The data-flow diagram (Figure 2.14) illustrates the component-level architecture of the motion stack, where blocks in the diagram stand for software components. Exception are blocks denoted by z^{-1}. These blocks are not part of the intended software system; they indicate the effects of communication (latency) modeled to verify robustness of the control algorithms as discussed in Section 2.6.2.

The elements of Orocos-RTT framework are classified using the 5C categories (Figure 2.16) :

- **Computation:** The code inside of the component;
- **Communication:** The inputs/outputs definitions, the connection policies;
- **Coordination:** The life-circle state machine;

- **Configuration:** The component properties;
- **Composition:** The connections between components.[11]

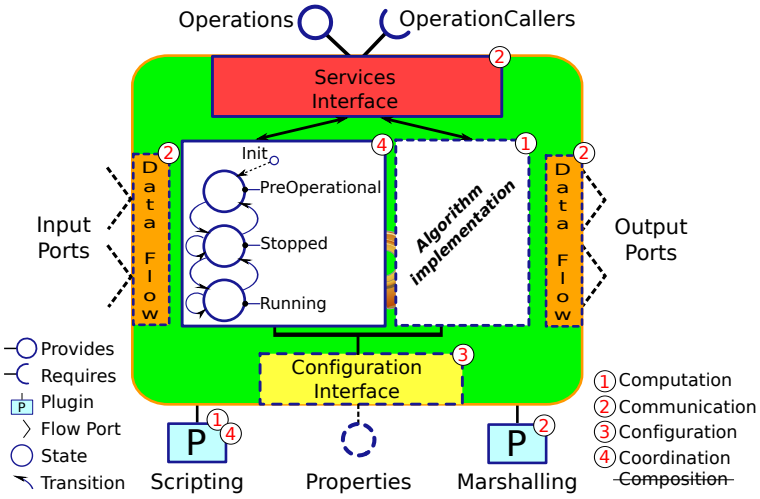


Figure 2.16: The model of a Orocos-RTT component (Orocos Project, 2013). The dashed elements are extracted from 20-sim, solid lines elements are parts of the template.

The block at the firmware and drivers level (denoted YouBotModel in Figure 2.14) implements the communication to the firmware of the youBot actuators and sensors. This component contains operating-system specific and hardware-specific code, and is written in a general purpose language (C++). This component is developed based on the original RAP development process. The component Configuration and Communication were explicitly modeled using BRIDE, while the other perspectives are implicitly modeled in the implementation code.

The blocks at the "trajectory planner" level of the given application (BRICkS stacking) are the Coordination-type components. The Coordination-type components are most efficiently expressed in a FSM, therefore a DSL for a FSM was used to design the actual trajectory planner components (Klotzbucher and Bruyninckx, 2007). The existing life-circle state machine of an Orocos-RTT component has been extended to include states required for the application. The Coordination was modeled using rFSM (Klotzbucher and Bruyninckx, 2007), a textual modeling language for FSM. The component Configuration and Communication were modeled using BRIDE. The model designed with rFSM is part of the working system, it is executed using a run-time interpreter based on Lua.

Other blocks in the data-flow diagram (Figure 2.14) are 'pure' Computation compo-

[11] Orocos-RTT at moment of development only offers flat composition of components, *i.e.* no hierarchical structures are supported.

nents. These components are implemented using the model-to-component transformation described in Section 2.5.

The model-to-component transformation requires a template that is used to transform modeling primitives into elements of the framework. The 20-sim modeling primitives map to elements of Orocos-RTT framework as follows (Figure 2.16):

- **Computation:** The code inside a component is generated from a chosen 20-sim submodel.
- **Communication:** The inputs/outputs of the model become the interface of the component.
- **Coordination:** Orocos-RTT defines a minimal life-circle state machine. It covers all necessary states: initial, configured, stopped, running, error, failure. This state-machine is sufficient for single non-coordination components, thus it has been set as part of the template.
- **Configuration:** The parameters of the 20-sim submodel is translated into properties in Orocos-RTT.
- **Composition:** The composition is not part of the exported Computation model.

The generated component is imported into BRIDE, where Communication and Configuration perspectives can be further refined to connect generated component into the system. Detailed instructions on using code generation process are presented by Brodskiy et al. (2013).

A composition model of the application is finalized using BRIDE. The models of computation are transformed into component models usable by BRIDE. These models are combined with the models components that were developed in BRIDE to obtain composition model. The run-time architecture of this use case is shown in Figure 2.17[12]. Due to the large number of components the image becomes very cluttered. For clarity three sets of components are marked: the motion stack, the driver and the application specific components. As can be seen, the application components are added to the motion stack and driver components, and complete the robotic application.

2.6.5 Results

The software integration developed in the section 2.5 is available as open source code (Wilterdink et al., 2013a,b) and is being used in other projects such as for example (de Boer et al., 2012). The use case was used to demonstrate the application of the proposed development process with support of the developed tool-chain. The resulted software (youBot motion stack) has been tested for ease of re-usability and reliability. For that 3 events of the BRICS project were used.

In the first event, the BRICS research Camp 3 (BRICS, 2011), the youBot motion stack was used to test re-usability of the components developed using the proposed

[12] Created with BRIDE version 02-08-2012 and a few minor adjustments

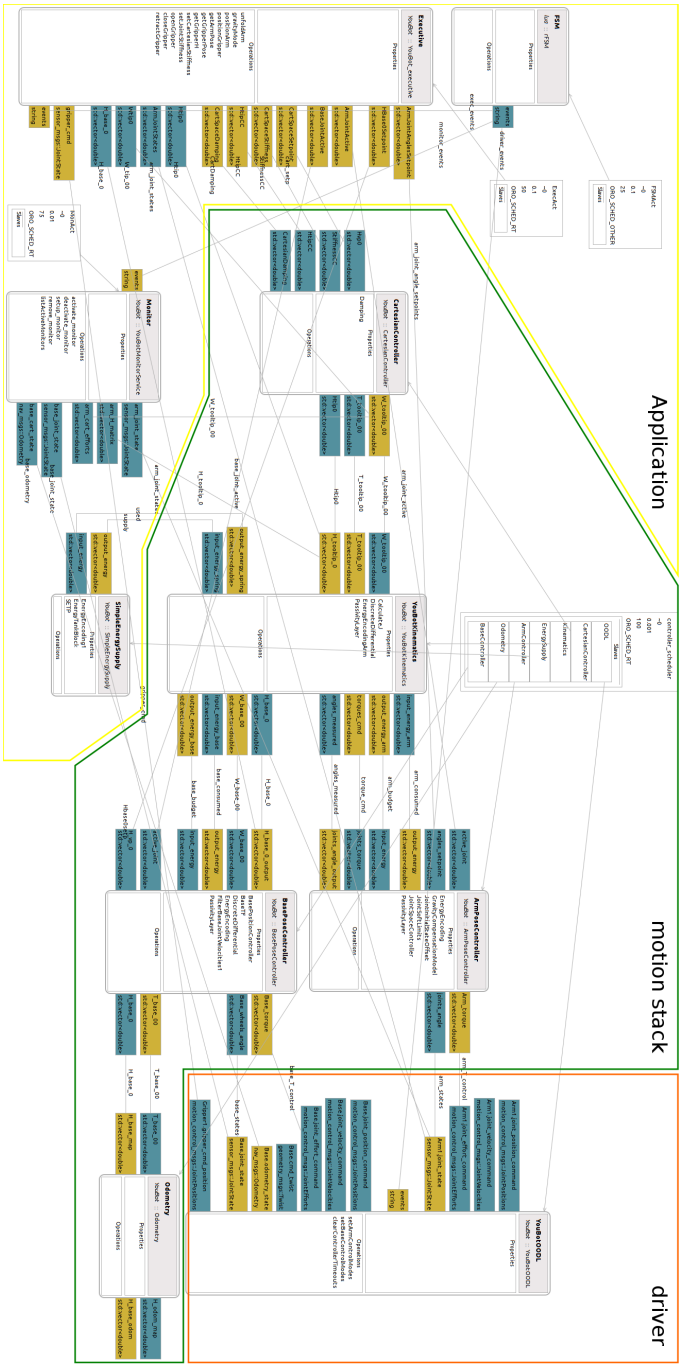


Figure 2.17: BRIDE architecture/composition model for RC3.

methodology. The software was provided to students, research camp participants, with only brief explanation of the algorithmic aspects of the system; the Computation and Coordination model were presented. The students have successfully used concepts of modeling of Computation and Coordination to adapt the provided system to their tasks, which demonstrates ease of re-use of the developed components.

In the second event, Automatica 2012 (BRICS, 2012a), the youBot motion stack was used to test reliability of the components developed using the proposed methodology. The youBot motion stack was used to demonstrate the reliability with respect to inter component communication faults. The developed algorithms were shown to work with communication over a congested WiFi (the detailed description of the algorithms is in Chapter 4). The software was subjected to prolonged active use, with frequent interruptions for inspection and parametric changes, which were done for demonstration purposes. The setup was performing flowlessly for whole week of the trade fair (5 days 8 hours a day), which indicates that Mean Time To Failure is longer than 40 hours.

To continue tests of re-usability and reliability of the the components developed using proposed methodology, the third event, the BRICS research Camp 5 (BRICS, 2012b) was used. The software has been given for modification to the BRSU RoboCup@work team. During this transfer, the team has been given an introduction to the software on the level of computation similar to Research camp 3 student teams. The RoboCup team has successfully modified parts of the motion stack and uses this software for robot control during completions. This also confirms that components developed using the proposed methodology can be easily re-use for different applications with high level of reliability.

2.7 Conclusions

The CBSD method (RAP) advocated by BRICS has been successfully combined with the ECS design trajectory. RAP is focused on re-use of off-the-shelf components, whereby the component boundary serves as an integration point. The ECS trajectory focuses on the design of algorithms by employing modeling and simulation techniques. The combination of both approaches results in uniform coverage of modeling perspectives (5C) for a software component, and thus results in more reliable robotic components and applications.

Based on the discussion presented, the proposed approach contributes to software quality improvement as follows:

- Automated model-to-code transformations reduce faults during implementation of the algorithm by excluding human factor from the process.
- Modeling enables the designer to focus on the chosen aspects of the system (*e.g.* algorithm or architecture), instead of implementation details, and thereby increasing quality of the resulted software.
- Simulation of the algorithm allows to examine hypothetical/dangerous situa-

tions using fault modeling techniques, such as sensor failures, which can be used for development of fault tolerance algorithms.

The experience with the implementation of the use case has shown that algorithms verified in simulation have a high success rate on a real setup, which confirms results reported by Broenink et al. (2010).

To achieve the uniform coverage of modeling perspectives, two different modeling tools had to be used. Each tool focuses on a different engineering role. Two tools were used as an example of the approach, BRIDE and 20-Sim. BRIDE is used to model architecture of the system, and 20-sim is used to model the algorithm.

The presented model-based tool integration shows that tools can easily be combined on the component level. The available model-to-text generator of the 20-Sim tool is used to generate an executable component. A model-to-model transformation is used to export the component interface of a 20-Sim sub-model to an Orocos-RTT component model. The Orocos-RTT component model is used in BRIDE to design the deployment of a robotic application. The advantage of directly generating the executable component from 20-Sim is that the target component meta-model is not required to support an equivalent Computation meta-model because the algorithm code is directly generated for the target framework. The integration of these tools provides an example of combining tools with two different meta-models to achieve modeling of the software from different perspectives. The integration approach is generic such that other tools can be integrated in a similar way.

3

Fault-tolerant control of mobile manipulators

Failure of hardware is an inevitable event in a long-term autonomous operation; therefore, the ability to detect and react to such failures is essential for an autonomous robot. Failures of sensors in mobile manipulators is a threat to safety for the environment and of the robot itself, which can be addressed by employing fault-tolerant control.

In this chapter, an approach for increasing the resilience of a mobile-manipulator to sensors failure is discussed. The proposed solution includes (a) a model-based fault detection and isolation algorithm to obtain fault signalling and (b) behavioral reactive control that enables the manipulator to continue execution. A model of a mobile manipulator (youBot) is considered to illustrate the approach and simulation results are presented.

3.1 Introduction

A distinguishing characteristic of robotic systems is the ability to affect the physical world. Sensors and actuators are elements of the system enabling that ability. For a robotic manipulator, the typical failures, which occur at loop-control level and jeopardize the safety of the robot, are malfunctioning sensors. Visinsky et al. (1994) demonstrated that, with position control, a robot will react on a sensor malfunction with immediate acceleration towards unsafe velocities and forces. This is a real problem for mobile robots since there are no safety barriers in an open environment.

A possible solution to improve safety and increase robustness of a robot is to improve the robot's fault tolerance. Fault-tolerant control is an essential element of fault-tolerant systems, because it defines the reaction on the failure of its elements. Since size and power consumption in mobile robots lead to minimizing the number of installed sensors and actuators, the goal of the fault-tolerant control for a mobile manipulator is to maximize the robustness of the robot with respect to failures based on existing sensors and actuators.

Fault-tolerant control is a type of control that takes the possibilities of structural or abrupt parametric changes in the controlled process into account, and adapts to these changes. As a hardware failure in a robot leads to immediate violent behavior, the speed of reaction is one of the most important characteristics for such type of control.

The structure of a fault-tolerant controller is presented in Figure 3.1. Fault Detection and Isolation (FDI) methods are used to identify possible failures of robot hardware and generate a failure hypothesis. A recovery procedure uses this failure hypothesis to modify the control law to minimize effect of the fault on the robot's performance.

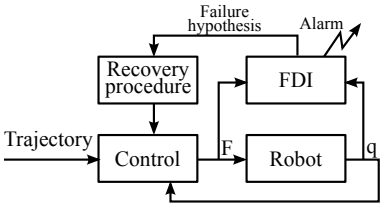


Figure 3.1: Architecture of a fault-tolerant control (FDI: Fault Detection and Isolation)

The blocks in Figure 3.1 are discussed in the rest of the chapter. An overview on FDI methods is discussed in Section 3.2. Recovery procedures are discussed in Section 3.3. The application of the discussed algorithms to youBot is presented in Section 3.4, which includes: a bond-graph-observer based FDI algorithm, the compliance control for youBot as single kinematic chain and the reconfigurations applied to the controller in case of hardware faults. The simulation is used to assess performance of the algorithms. The chapter is concluded by Section 3.5.

3.2 Fault Detection and Isolation

An FDI algorithm generates a failure hypothesis based on known inputs, measurements and a priori knowledge of the system. To generate a failure hypothesis such algorithms should identify the existence of the failure in the monitored process and identify the cause of the failure. Identification of the fault existence is termed fault detection, while identification of the failure cause is termed fault isolation or fault diagnosis (Isermann, 2006). An FDI algorithm is composed of a combination of several signal processing algorithms. A review of the various algorithms employed to solve the FDI problem, the characteristic used to evaluate their combined performance and the applicability to mobile robots is presented in this section.

3.2.1 Architecture of an FDI algorithm

An FDI algorithm performs a transformation from available data to a failure hypothesis. FDI methods, in general, are structured as a series of transformations according to Figure 3.2 adopted from (Isermann, 2006; Venkatasubramanian et al., 2003). Terms in brackets proposed by Venkatasubramanian et al. (2003).

The presented architecture indicates that three types of algorithms are required to produce a failure hypothesis. The detection of a hardware fault and isolation of a failed element requires an algorithm that possess a representation of the monitored hardware. There are two sources from which information about the monitored system can be obtained, a system model or the system's history (Venkatasubramanian et al., 2003).

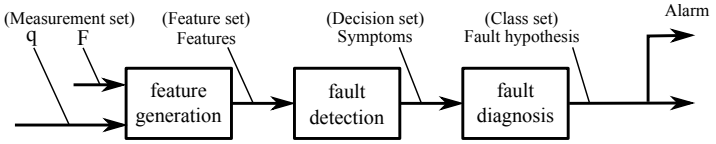


Figure 3.2: Transformations in a FDI system.

It is convenient to classify the algorithms used in an FDI by the type of information they use and their purpose in the FDI structure (blocks in Figure 3.2). Such classification allows to choose an appropriate algorithm depending on the information type available to a developer.

Feature Generation

The measurement set (Figure 3.2) is formed by all possible measurement and input vectors. The information redundancy about the monitored system is used to develop feature generation (a mapping between the measurement set and a feature set). The classification of the feature-generation algorithms, based on the type of information used, is illustrated in Figure 3.3.

- **Model-based methods** use explicit models as a source of informational redundancy. These methods are developed based on understanding of the monitored system. The model is explicitly presented as a part of the algorithm or was used to develop the desired transformations.

- **Process-history-based methods** rely on stored information and experience of past behaviors of the system. In contrast to the model-based methods which use an explicit model of the system, FDI based on a process-history method uses an implicit model of the process, inferred from information about the past operation of the system.

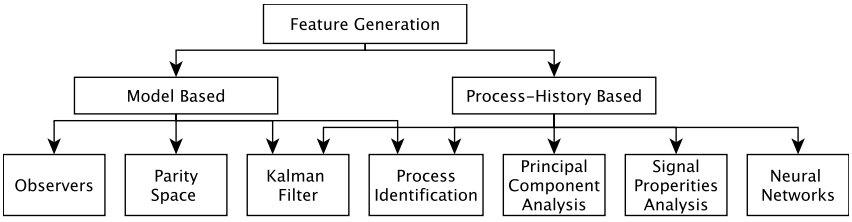
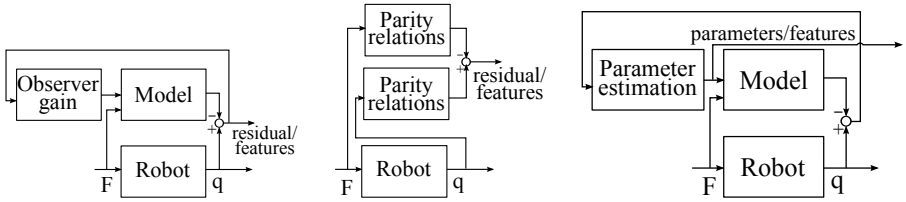


Figure 3.3: Classification of feature generation algorithms

Observers-based methods are the most widely used type of feature generation (Isermann, 1997). The observer is used to predict the measurement vector, while the difference between estimated and measured values (residuals) is used as features to detect faults. A distinguishing structural property of an observers-based feature generation

is the use of the residuals to correct the state estimation. Thereby, an observer-based feature generation has a structure with a feedback loop as illustrated in Figure 3.4a.

The *parity space* approach is similar to the observer approach, but the model of the monitored system is structured differently (Christophe et al., 2004). The model is used to define quantitative relations between inputs, sensor measurements and their derivatives. The parity space method also produces features in a form of inconsistency between the expected and measured states. In the parity space method, the residuals are not used to correct the state estimation, but the relations between states are defined over a time period. Thereby a party space feature generation has a structure without a feedback as illustrated in Figure 3.4b.



(a) Observer-based **(b)** Parity space-based **(c)** Process identification-based

Figure 3.4: Architecture of the feature generation

The *Kalman filter based* methods are a set of algorithms that can be used for both estimation of states and parameters of the monitored system (Welch and Bishop, 2006). For state estimation, Kalman filter feature generation works similarly to observer-based methods. When it is used for parameter estimation, it is similar to process identification methods.

The *Kalman filter* and *process identification* approach are hybrid approaches. The process identification approach produces features in the form of parameters of the monitored system or their deviations (Isermann, 1993). These methods contain a structural model of the monitored system while parametric properties are determined using process history data. The structure of process identification method is depicted in Figure 3.4c.

Principal component analysis (PCA) can be used for run-time monitoring of a system, it is used to produce residuals based on expected and measured states (Nomikos and MacGregor, 1994). The process history, which is represented by a record of the measurements and inputs of the correctly working system, is analysed to determine principle components. The principle components are latent variables that are linearly independent from each other and account for as much of the variability in the analysed data as possible for a given number of components. The principle components are used to define a correlation between elements of the measurement set. The violation of this correlation is used to produce the residuals. The architecture of a PCA based feature generation is depicted in Figure 3.5.

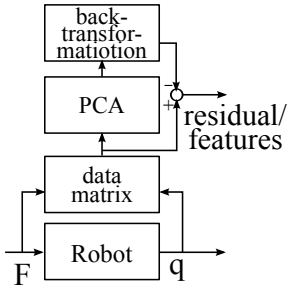


Figure 3.5: Architecture of PCA based feature generation (Isermann, 2006)

The *signal properties* analysis is employed for detecting a change in periodic signals. This type of methods produce features such as change in signal spectrum, correlation, amplitude or power density.

The *neural networks* approach can be used as a complete FDI process or any of its sub algorithms. The process history used in the *neural network* approach combines records of the measurements and inputs of the correctly working system, and the system with failed components. This process history is used as input for training the neural network. The trained neural network can be used as an observer to predict the state of the monitored system; this prediction is being used to generate residuals. The trained neural network can also be used to directly produce a fault class from the measurements, features or symptoms.

The above-mentioned pure process-history-based methods are used in situations where little or no information is available about structure or parameters of the monitored system, or in situations where modeling the process is economically infeasible.

On the other hand, the model-based methods are well suited for fault detection in the systems described by known physical laws with most of their parameters available such as in robots (Isermann, 2006). The observer type feature generation has an advantage of simplicity of implementation. Moreover, a variety of observer methods have been developed for a class of non-linear systems (Isermann, 2006; Abid, 2010; Garcia and Frank, 1997; Garg and Hedrick, 1995; Thau, 1973) similar to a typical mobile manipulators.

The residual generation for a mobile manipulator, for example, can be obtained by using a Thau observer. A Thau observer allows to construct an observer for a non-linear system with a constant observer gain, if the observation function is linear and the type of non-linearity is known (Thau, 1973; Rajamani, 1998). The structure of the Thau observer is given as:

$$\begin{aligned} \dot{\hat{x}}(t) &= A\hat{x}(t) + g(\hat{x}(t), u(t)) + L(y - \hat{y}) \\ \hat{y}(t) &= C\hat{x}(t) \\ r(t) &= y - \hat{y}(t) \end{aligned} \tag{3.1}$$

where

- $A\hat{x}(t) + g(\hat{x}(t), u(t))$ is the model of dynamics of the observed system;
- C is the observation matrix,L is the observer gain;
- $\hat{x}$ is the vector of the observer states;
- y is the vector of measured values;
- $\hat{y}$ is the vector of estimation of the measured values;
- r is the the vector of residuals used for fault detection.

Fault Detection

The features (Figure 3.2) are transformed into symptoms using fault detection algorithms. The classification of fault detection algorithms is based on the type of information used to construct them as illustrated in Figure 3.6. This mapping of the feature set into the decision set is the point of engineering trade off between characteristics of the FDI method. The detection algorithms are typically thresholds, which are used to determine significant change in the features.

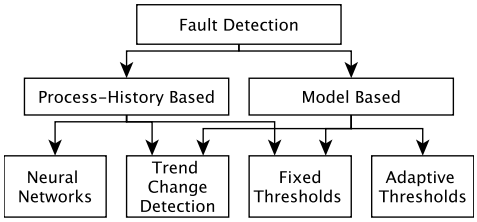


Figure 3.6: The classification of fault detection methods

Thresholds are determined based on the level of noise, tolerable changes in the monitored system or modeling errors, using statistical and model based analysis (Ding, 2008). A threshold provides a binary output signal, indicating that a feature has exceeded its allowed range. If the derivatives of features are used, the method can be classified as *trend-change detection*. Adaptive thresholds are used to avoid false fault detection due to modeling errors.

An *adaptive threshold* changes the allowed range of the monitored feature based on the expected dynamics of the monitored system. Modeling errors lead to non-zero residuals with dependency on the system dynamics and more specifically on input signals. By increasing the allowed range of the residual based on current control signal, false detection is avoided. An example of adaptive threshold (illustrated in Figure 3.7) can be used to avoid false fault detection by increasing the threshold depending on the input (control) signal. A high pass filter (HP.F) is used to enlarge the threshold to compensate for modeling errors in the dynamics of the system. C_2 is used to compensate for linear dependencies between input signal and the residuals, and C_1 is a

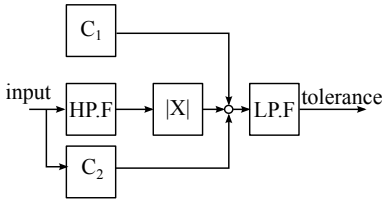


Figure 3.7: An example of adaptive threshold based on the control input (Isermann, 2006)

static part of the threshold, while the low-pass filter (LP.F) is used to smooth changes in the threshold.

Fault Diagnosis

The fault-diagnosis block transforms symptoms into a fault hypothesis (Figure 3.2). Fault-diagnosis algorithms are typical classification and clustering algorithms. These algorithms can be classified based on the type of information used to construct them (Figure 3.8). Model based methods rely on structural or casual properties of the monitored system to identify the cause of the symptoms, while process history based methods use knowledge about the system performance, such as experience of operators or maintenance crew.

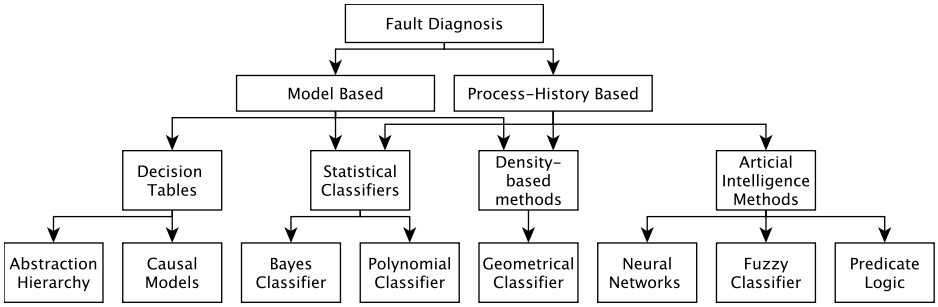


Figure 3.8: Classification of fault-diagnosis methods

Decision tables represent the simplest way to diagnose faults. The table can be constructed based on a casual model or an abstraction hierarchy (functional or structural composition of the system). The *decision tables* are typically combined with a feature-generation method that provides changes in the measured states or system parameters as features

Statistical classifiers and *density-based methods* are used when the structure of the symptom space is known and too complex to be described by decision tables. To structure the symptom space, assumptions about types and probabilistic occurrences of faults have to be taken, this requires both model and history process information.

Artificial intelligence methods use inference to identify the structure of the symptom space. The structure of the monitored system and rules that link symptoms to a possible cause of failure are provided. Given a structure of the monitored system and rules that link symptoms to possible changes in the system, an inference algorithm searches for the most probable cause of a failure based on a given symptom. In *artificial intelligence* methods, there is no explicit link between symptoms and failure cause, it is computed by the algorithm based on the provided knowledge of the system.

Means to obtain information for fault diagnosis method are based on fault forecasting approaches. A number of techniques were developed to itemize, assess the probability of occurrence and the system reaction on faults. Fault forecasting methods rely on:

- Scenario based analysis - Failure mode effect and criticality analysis (FMECA) (USA Department Of Defense, 1980), Software Architecture Reliability Analysis approacH (SARAH) (Sözer, 2009)
- Cause and effect analysis - Fault Tree Analysis (FTA), Event Tree Analysis (ETA)
- Risk assessments - Stress Strength Analysis (Blanchard and Fabrycky, 2006), Reliability prediction (USA Department Of Defense, 1991)

For example, FTA is a top down approach to construct the fault class set, involving graphical enumeration and analysis of ways in which a failure can occur. The main part of the FTA is construction of the tree of events in which a parent node is a consequence of a child node. A root of the fault tree is a top level event which is a clearly observable failure of a system to perform a specified function. If the system has more than one core function, multiple fault trees are constructed. The possible causes of a failure are listed as leaves of the fault tree. Each leaf is further expanded to identify the cause of that fault by generating the cause hypothesises for it. The FTA is stopped when the desired granularity of fault causes is achieved. The terminating leaves of the fault tree comprise the fault class set. The FTA can be graphically represented using logical blocks, as described for example by (Blanchard and Fabrycky, 2006) and illustrated in Figure 3.9.

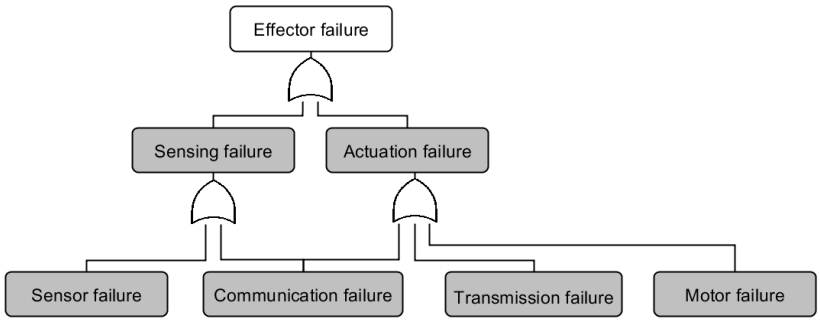


Figure 3.9: FTA of an effector

FTA for effector failure as was defined in Chapter 1, is depicted in Figure 3.9. The links have negligible probability of destruction with a prescribed workload whereas the Mean Time Between Failure (MTBF) for a joint is in the range of 5000 hours (Groom et al., 1999). Similarly to systematic analysis for manipulator failures reported by Visinsky (1991), the FTA yields relations between possible faults presented in Figure 3.9. The boxes in Figure 3.9 indicate the elements of the fault class set.

3.2.2 Characteristics of an FDI algorithm

To evaluate the quality of the transformation of sensory data to a failure hypothesis several properties of an FDI should be highlighted. Based on the analysis of FDI systems in chemical process engineering by Venkatasubramanian et al. (2003), the properties applicable to an autonomous robot can be summarised as follows:

- **The response time** is characterised by a time delay between a failure event and the system response. It is the delay between deviation of the process parameters and detection of this deviation. The minimum response time is determined by the dynamics of the system and by the consequences of the failure. Filtering and thresholds, added to the system to avoid false detection, increase this delay.

- **Robustness** is characterised by the amount of additive noise in the sensory data that the FDI system can tolerate while maintaining the specified probability of false detections (false positives).

- **Isolability** is the ability to identify a system part in which a fault has occurred. It can be measured as a percentage of the system that is considered faulty upon a given event. Better isolability implies that the fault hypothesis, produced by a given FDI method, will deem less elements of the system as faulty ones.

- **Novelty identifiability** is the ability of a system to detect new (unknown) faults that were not defined at design time. If a system is constructed in a structured way with attention to fault identification at every level of abstraction and for every component, it could isolate failed elements without a priori knowledge of the cause. This metrics can be measured in the same way as isolability but for new (unknown) faults.

- **Multiple identifiability** characterises how many simultaneously occurring faults a system can recognise.

- **The Correctness and classification probability estimate** describes the probability of a correct detection to a certain fault. An unstructured and noisy environment can affect the identification and isolation processes such that the FDI algorithm is able to detect irregularity in behaviour of monitored system, but is unable to classify it properly.

From these FDI properties, the response time, robustness and isolability are relevant for deterministic model-based fault detection methods, which is used in our use case. The novelty identifiability and the correctness and classification probability estimate

are used to determine the quality of the fault diagnosis for process history based methods.

3.3 Recovery from faults

The purpose of the recovery procedure is to reduce the effect of a hardware failure on the performance of the main task. The recovery procedures in mobile robots are focused on fault tolerance, as no recovery procedure will restore the hardware autonomously to normal operation. There are two types of fault tolerance possible: compensation and degradation. The compensation uses redundant elements of the system to completely mask the presence of a fault. Degradation involves a partial compensation and implies a loss of some performance with respect to the main task.

Full compensation of faults requires a redundancy that can be used to completely take over the function of the failed hardware. Typical hardware redundancy involves a replicate of the hardware elements (stand-by hardware). The recovery procedure of this type of redundancy is switch between active hardware element to a stand-by one.

Partial compensation and degradation relies on hardware elements that perform similar functions. The partial compensation is possible if the task of the failed element can be redistributed among several other operational hardware elements. The task and available hardware dictate how the degradation can be performed.

3.3.1 Recovery procedures specific for mobile manipulators

The purpose of the recovery procedure for a manipulator is to maximize its ability to continue execution of its task, to affect its environment in a planed and controller manner. As was noted above, the typical types of failures for a manipulator are sensing and actuation failures, therefore the recovery procedures for these types of failures will provide maximum impact on robustness of a manipulator.

Kinematic redundancy (as defined in Conkur and Buckingham (1997)) is a specific type of hardware redundancy often available in manipulators and mobile manipulators. The original purpose of this type of redundancy is to increase the number of types of the tasks the robot can perform. Nevertheless, this type of redundancy can also be exploited to compensate for faults in robot arm actuation and sensing if the main task does not require the use of all available degrees of freedom. The goal of the recovery procedure, in this case, is to maximize this reachable space, as bigger reachable space increases robot's ability to perform manipulation. The recovery procedure that allows to maximize reachable space of a robot despite hardware failures by using existing kinematic redundancy is specific for manipulators and mobile manipulators; this idea was utilized on the example of the youBot.

Fault-tolerant systems can be categorized into three groups based on their fault-tolerant capabilities. With respect to the desired recovery procedure, on the example of the robot arm joint actuators, 3 types of fault-tolerant systems are defined as:

- **No fault tolerance**
  If a robot has an actuator without fault-tolerant features in an open kinematic

chain, failure of this actuator can not be efficiently compensated a due to uncontrolled and un-measurable motion. The only recovery option is to power down the actuator, which in most cases significantly reduces the working envelope and results the swing of the links connected to the broken joint due to gravity or interaction forces. A typical recovery solution is to interrupt the task to ensure fail-safe operation of the complete robot.

- **Fail safe**
  A robot with fail-safe actuators has breaks to prevent a joint from moving after a failure has occurred. The breaks are cold stand-by actuation that is only applied when a joint failure is detected. Engaging the breaks makes the joint equivalent to a rigid connection, thus the robot can continue the task if the controller is properly reconfigured. For parallel kinematic structures (closed kinematic chains or kinematic loops) fail-safe actuation does not require breaks, as the use of breaks would interfere with actuation. The load of the broken joint can be distributed between actuators within the structure; the failed actuator should allow free motion of connected links.

- **Fail operational**
  A robot with fail-operational actuators has a redundant driving units, such that the task can be continued without interruption. The reconfiguration process in such case is simple switching between active and stand-by actuation.

Note, that application of fault-tolerant control allows to increase system resilience to the faults by one step. In other words, a robot with fail-safe actuators can perform as fail-operational system or a robot with no fault tolerance actuators will become a fail-safe system. The exact controller reconfiguration depends on the control strategy.

3.4 Use case: fault-tolerant control for youBot

A mobile manipulator (youBot) has been used to demonstrate the use of fault-tolerant control in combination with compliance control.

The youBot (Figure 3.10a) is a robot produced by KUKA for experiments/research in control and software engineering (Bischoff et al., 2011). It consist of an arm mounted on a mobile omni-direction base. The youBot arm has six links connected by 5 actuated rotational joints. The axis of rotation for all arm joints is the z-axis in the frames depicted in Figure 3.10b (the positive direction of the z-axes is determined by KUKA convention (Locomotec, 2010)). Four Mecanum wheels are mounted to the first link (base), see the schema in Figure 3.14. A detailed platform description and dynamic model are presented in Appendix A.

The architecture of the fault-tolerant control for youBot is presented in Figure 3.11. Compared to the architecture of the fault-tolerant control in Figure 3.1, there are 3 additional blocks. The *Quality_measures* block is used to determine performance during trajectory tracking by computing distance between desired and actual position of the tool-tip. Blocks *Failure motors* and *Failure sensors* are used to model the faults. Furthermore, the connection between *youBot_model* and *Controller* is a bond (bond

graph notation), thus the observer in this architecture acts in two roles: the robot state estimation as part of the controller feedback loop and residual generation as part of FDI.

3.4.1 FDI for youBot

According to Figure 3.2 three transformations have to be defined within the FDI algorithm to obtain a failure hypothesis from measurement data.

The first transformation is generation of the features. A proiri information is the model of the youBot which provides structural and parametric description of the system to be monitored. Therefore, an observer-based feature generation is chosen. A Thau observer presented in section 3.2 (Eq. 3.1) is employed to avoid on-line computation of the observer gain. In Figure 3.11 observer based feature generation is indicated by two blocks *youBot_model* and *Observer_gain*. The additional signal into *Observer_gain* coming from the *Controller* block is needed to compensate for changes due to recovery procedure. A bond-graph model of the youBot (presented in the Appendix A) is used as the model of the dynamics of the observed system (Eq. 3.1).

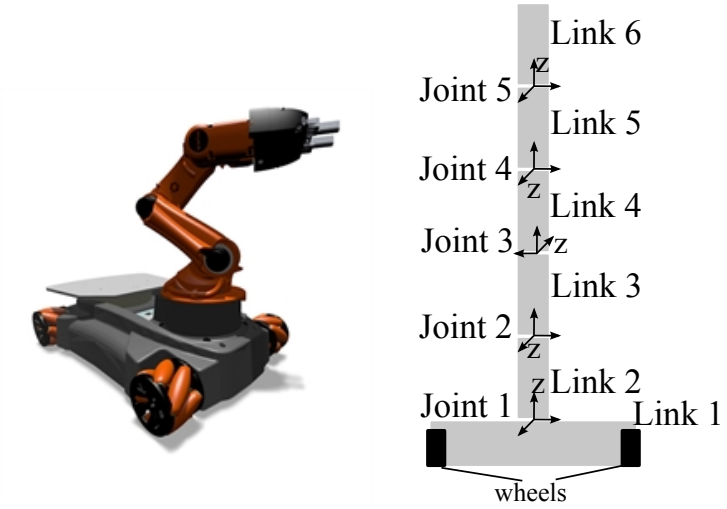
The second transformation is the detection of the faults. Static thresholds are used for that purpose. The outputs of the thresholds are the symptoms used for the fault diagnosis.

The third transformation is the fault detection. A decision table based on causal model is used. The FTA presented in Figure 3.9 is the adopted causal model. In case of the youBot, a single position-sensor per actuator is available, which constitutes 9 independent measurements, thus using a single observer based feature generation only 9 fault types can be isolated (White and Speyer, 1987). The maximum level of isolation is chosen as failure of the "effector failure" (Figure 3.9) to be able to identify failure in all 9 actuators. More specifically, it is not possible to distinguish between failures of a sensor or motor drive of the same effector.

3.4.2 Control algorithm

The youBot has back-drivable actuators with low friction and position feedback. The actuators are controlled via current to the motors. Therefore, the youBot fulfills the conditions required for the application of an impedance control as described by Stramigioli and Bruyninckx (2001); Stramigioli (2001). Stramigioli and Bruyninckx (2001); Colgate and Hogan (1989, 1988, 1987) have demonstrated several advantages of this control law:

- The stability of such a controller can be guaranteed despite modeling uncertainties of the controlled robot and interaction with unstructured environment.
- A naturally robust controller can be achieved.
- Compliant behavior of the robot can be achieved if the spring elements are dimensioned properly.
- There is a physical interpretation of the control law.



(a) Appearance **(b)** Schema of the youBot, front view

Figure 3.10: The youBot

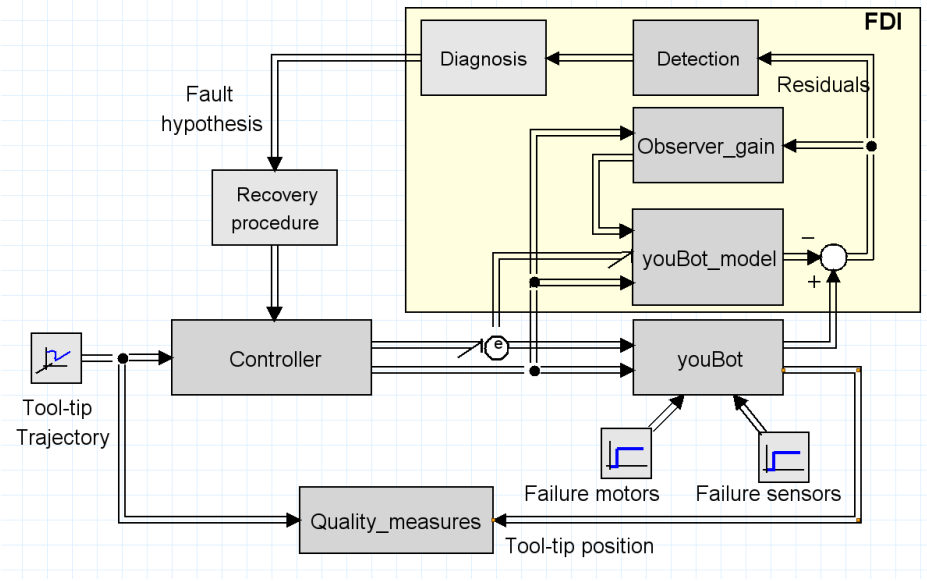


Figure 3.11: Signal flow diagram of simulation to study of Fault-Tolerant Control for youBot

This control law defines a relation between the robot's configuration and the forces that the robot exerts on the environment. This relation is defined by an artificial potential field, which is designed according to its task (Ott et al., 2005; Bäuml et al., 2010; Albu-Schaffer et al., 2007).

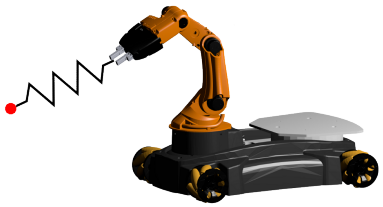


Figure 3.12: The control strategy for the youBot

The control strategy for the tool tip used in this case study is based on a Cartesian stiffness using a spatial spring definition (Stramigioli, 1998, p. 167). The control strategy is intuitively depicted in Figure 3.12. A virtual spatial spring is used to define a minimum of potential energy, which coincide with desired configuration of the robot (the tool-tip is in the set-point). The generalized force (wrench $W_{tt}^{0,0}$) is computed based on a spatial spring definition (Eq. 3.2).

$$
\begin{aligned}
W_{tt}^{0,0} &= \left[(m_{tt}^{0,0})^T \quad (f_{tt}^{0,0})^T \right]^T \\
\widetilde{m}_{tt}^{0,0} &= -2 * \mathrm{as}(G_o R_{tt}^{vp}) - \mathrm{as}(G_t R_{vp}^{tt} \widetilde{p}_{tt}^{vp} \widetilde{p}_{tt}^{vp} R_{tt}^{vp}) - 2\mathrm{as}(G_c \widetilde{p}_{tt}^{vp} R_{tt}^{vp}) \\
\widetilde{f}_{tt}^{0,0} &= -R_{vp}^{tt} \mathrm{as}(G_t \widetilde{p}_{tt}^{vp}) R_{tt}^{vp} - \mathrm{as}(G_t R_{vp}^{tt} \widetilde{p}_{tt}^{vp} R_{tt}^{vp}) - 2\mathrm{as}(G_c R_{tt}^{vp})
\end{aligned}
\tag{3.2}
$$

where

- $m_{tt}^{0,0}$ is the rotational part of the generalized force;
- $f_{tt}^{0,0}$ is the translational part of the generalized force;
- tt is the tool tip of the robot;
- vp is the virtual position representing the desired position and orientation of tool tip;
- G_o, G_t, G_c are the co-stiffness matrices obtained form input stiffness matrices as $G_x = \mathrm{tr}(K_x) I - K_x, \quad x \in \{o, t, c\}$, where the physical interpretations of the orientation, translation and coupling stiffness matrices are given for example by Loncaric (1987);
- R_i^j is the orientation matrix between two coordinate frames i, j;
- p_i^j is the translation vector between two coordinate frames i, j;
- $\mathrm{as}(A) = (A - A^T)/2$ is the operator that returns the anti-symmetric matrix;

- $\widetilde{x}$ is the skew operator on the vector x.

The wrench defined by the virtual spring is projected on the tool tip ($W_{tt}^{0,0}$). In other words, the robot uses its actuators to produce a wrench equal to the one defined by the virtual spring, thus emulating the presence of the spring. This results in moving the tool tip towards the desired position.

The joint actuators are used to create the desired wrench on the tool tip of the robot. To obtain the required joint torques, the Cartesian space wrench has to be mapped onto joint-space torques. To make use of existing kinematic redundancy, the arm and the base have to be used as single kinematic chain. This is achieved by two stage mapping (Figure 3.13):

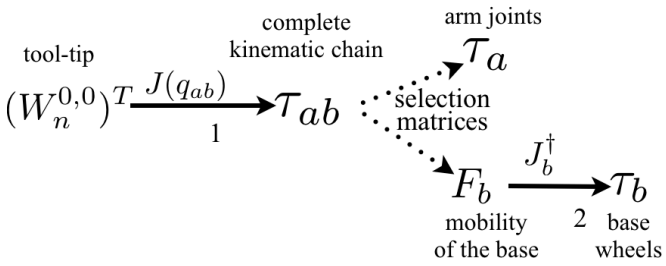


Figure 3.13: Mapping between the wrench on the tool tip and joint torques

1. A geometric Jacobian ($J_{ab}^T(q)$) is used to map the Cartesian space wrench ($W_{tt}^{0,0}$) to joint-space torques (Nakamura, 1990, p. 113). In this joint space the mobility of the robot's base is represented by a virtual joint with three degrees of freedom: translation along the x-axis of the base, translation along the y-axis of the base and rotation around the z-axis of the base (shown as (x, y, α) in Figure 3.14). Thus the joint-space is 8-dimensional, five belong to the joints of the arm and three to the base.

   $$\tau_{ab} = J_{ab}^T(q_{ab}) W_{tt}^{0,0} \tag{3.3}$$

   where q_{ab} is the 8-dimensional vector of states of virtual joint of the base and the joints of the arm; $\tau_{ab} = [F_b^T; \tau_a^T]^T$ is the combined vector of forces on virtual joint of the base and the torques on the joints of the arm. τ_a are the toques that are commanded to the joint motors of the arm.

2. The forces of the virtual joint of the base (F_b) have to be mapped onto the torques of the base wheel motors (τ_b).

   The relation between force, created by the Mechanum 4-wheel base, and wheel torques is obtained based on non-slip-conditions and the kinematic configuration of the youBot base (Figure 3.14), as described by Siciliano and Khatib (2008, p. 394). The reaction forces normal to the floor are not represented in Eq. 3.4 as it is assumed that youBot is moving on flat surface.

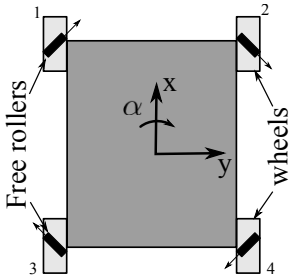


Figure 3.14: Schema of the youBot base. Small arrows on the rollers indicate the direction of the force applied by a wheel to a ground during rotation of the wheel.

$$F_b = \begin{bmatrix} F_\alpha \\ F_x \\ F_z \end{bmatrix} = \frac{1}{2r_w} \begin{bmatrix} x_r + y_r & x_r + y_r & x_r + y_r & x_r + y_r \\ -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \end{bmatrix} = J_b \tau_b \qquad (3.4)$$

where F_α is the torque created by the base about its geometric center, F_x and F_y are the forces along the x-axis and y-axis of the base (Figure 3.14), r_w is the wheel's radius, x_r and y_r are coordinates of the contact point of a wheel on the floor with respect to the geometric center of the base (the symmetry point), τ_b is the vector of the base wheel torques. The youBot base is symmetric and the positive directions of the wheel torques are defined using KUKA convention (Locomotec, 2010) and used here.

The relation between the forces of the virtual joint of the base and the torques on the base wheels is the inverse of Eq. 3.4. The Jacobian (J_b) is deficient in rank, thus there exist an infinite number of combinations of torques on the wheels (τ_b) that result in the desired virtual joint forces (F_b). A pseudo inverse for the Jacobian (J_b) has to be defined to find torques on the wheels (τ_b) based on the forces in virtual joints (F_b). Following Doty et al. (1993), definition of the pseudo inverse require physically consistent metrics to be defined for F_b and τ_b. To define a metric for F_b and τ_b, their conjugate variables have to be considered, which are the corresponding velocities of the base $V_b = [\dot{\alpha}, \dot{x}, \dot{y}]^T$ and the rotational velocities of the wheels $\dot{q}_b = [\dot{q}_1, \dot{q}_2, \dot{q}_3, \dot{q}_4]^T$. As V_b and $\dot{q}_b$ are conjugate to F_b and τ_b the following relation is true:

$$V_b = J_b^T \dot{q}_b \qquad (3.5)$$

In Eq. 3.5, only V_b has a physically consistent metric (M_V) which can be derived using kinetic energy. A metric for $\dot{q}_b$ can be defined such that a generalized inner product $\dot{q}_b \langle \bullet \rangle M_q \dot{q}_b$ made sense in physical units (Doty et al., 1993). $\dot{q}_b$ are rotational velocities on the identical motors, thus no scaling between units is required and M_q is chosen as identity matrix $I_{4\times4}$. Since J_b^T is full column rank, its generalized pseudo inverse does not depend on the metric M_V (Doty

et al., 1993). Thus Eq. 3.5 can be inverted as follows:

$$\dot{q}_b = [M_q^{1/2} J_b^T]^\dagger M_q^{1/2} V_b = [J_b^\dagger]^T V_b \tag{3.6}$$

where $J_b^\dagger = J_b^T (J_b J_b^T)^{-1}$), Moore-Penrose pseudo-inverse of the Jacobian. Finally, since Eq. 3.4 and 3.5 are dual , the torque on the wheels (τ_b) can be obtained as:

$$\tau_b = J_b^\dagger F_b \tag{3.7}$$

The resulting actuator-space $[\tau_a; \tau_b]$ is nine-dimensional, where five belong to the joints of the arm and four belong to the wheels of the base.

3.4.3 Recovery procedure for youBot

The final step in design of fault-tolerant control is to design a recovery action. As noted above (Section 3.3), kinematic redundancy can be used to compensate for joint failure. This allows to position the tool-tip and to perform manipulation tasks with a broken wheel-drive or joint-drive. The reconfiguration of the controller is required to exploit this idea.

The youBot's arm is an open kinematic chain, therefore, to maintain operation, the actuators have to be at least fail-safe. For the existing hardware, the only response possible in case of failure detection is stopping the operation. The reconfiguration process for the robot with fail-safe actuation is of most interest, therefore in simulations and in recovery procedure design the joints are considered fail-safe (have breaks).

As part of reconfiguration, the sensor readings of the failed joint should be fixed to the last correctly estimated measurement to determine the configuration of the kinematic chain. The control signal is set to zero. A joint with engaged breaks is indistinguishable from a rigid connection, thus two links connected by this joint can be considered as a new single link. The wrench defined on the tool tip ($W_{tt}^{0,0}$) has to be mapped to the joints torques (τ_a) using the changed kinematic structure. Due to the process of construction of the Geometric Jacobian (J_{ab}) a new mapping is obtained by removing the column that corresponds to the failed joint.

The youBot's base is a parallel structure and can be considered as a kinematic loop. Thus, fail-safe operation of the wheel-drive is free rotation of the wheel. Holding a wheel with a brake will impair mobility of the robot by constraining motion of the wheel, due to the parallel structure of the mechanism for the base. During reconfiguration, the wheel actuator is powered down to minimize the friction induced by the uncontrollable wheel. The sensor information from this wheel is ignored, and the three sensors on the other wheels are used to compute the odometery. Actuation of 3 Mecanum wheel of the base allows omni-directional movement (Siciliano and Khatib, 2008, p. 167), completely preserving initial mobility of the base. The mapping between the force created by the base (F_b) and the forces applied by the Mecanum wheels (τ_b) can be computed by removing the column that corresponds to uncontrolled wheel from the base Jacobian (J_b) before computing the pseudo-inverse (since J_b becomes a full rank and the pseudo-inverse will be the normal inverse).

3.4.4 Simulation

The fault injection approach (as noted in Chapter 2) was taken to test the performance of the proposed fault detection and the recovery procedures. In simulation, the youBot had to perform a trajectory tracking with the tool-tip, while faults are introduced. Two types of simulation are recorded with stand-by breaks on joints of the youBot arm and without. In case of no breaks on the joints, the reconfiguration performs power down, instead of engaging the break.

Simulations for three types of failure are presented: failure of a sensor in the 2nd joint of the arm (Figure 3.10b), failure of a sensor in the 1st wheel-drive (Figure 3.14) and failure of the motor in the 3rd joint (Figure 3.10b). These failures have been chosen as incorrect reading from a joint further away form the tool tip results in a higher tracking error. To produce a sensor failure, the sensor value is set to be “frozen" (does not change). Such error in sensor behavior was indicated as being the highest probability sensor failure by Carlson et al. (2004). During actuator failure, the motor torque is set to zero to simulate failures such as loss of power or communication signal.

A single failure was injected at t=1.0 seconds after the start of performing the task, then the the youBot is followed for 40 seconds, during which it has to track a predefined trajectory. Prior to these simulations, the performance of the youBot without failures was recorded such that a possible performance drop can be assessed. The traces in time of the residual for the failing actuator, the youBot's base and the tool tip trajectories from the simulation are recorded.

3.4.5 Results

The trajectories of the youBot base during the task execution are presented in Figure 3.15. Three trajectories are shown, namely during nominal execution, after fault injection with out and with the proposed fault-tolerant control. With fault-tolerant control, the base follows a trajectory similar to the nominal case, while without it the robot moves in a different direction.

Figure 3.16 illustrates the projection of the tool-tip on the xy-plane (ground). Three trajectories are shown as in the previous figure. The vectors indicate the projection on xy-plane of the direction of the tool-tip. It is clearly seen that the fault-tolerant control (green line and vectors) allows to maintain the same trajectory and orientation of the tool-tip as in the nominal case (blue line and vectors).

The residual traces during failure of the wheel drive sensor are presented in Figure3.17, and the arm joint sensor failure is presented in 3.18. The traces of the residuals are used to show the response time (time between failure t=1.0 second and reaching threshold). The residuals reach the set threshold in 0.015 seconds and 0.22 seconds. The other residuals are not shown as those remain well below the threshold.

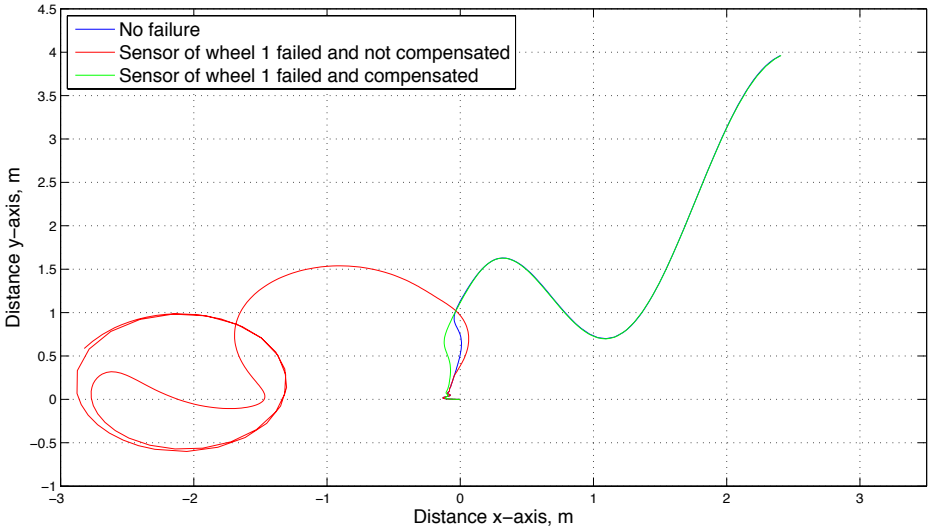


Figure 3.15: Trajectory of the youBot base during trajectory tracking with tool tip

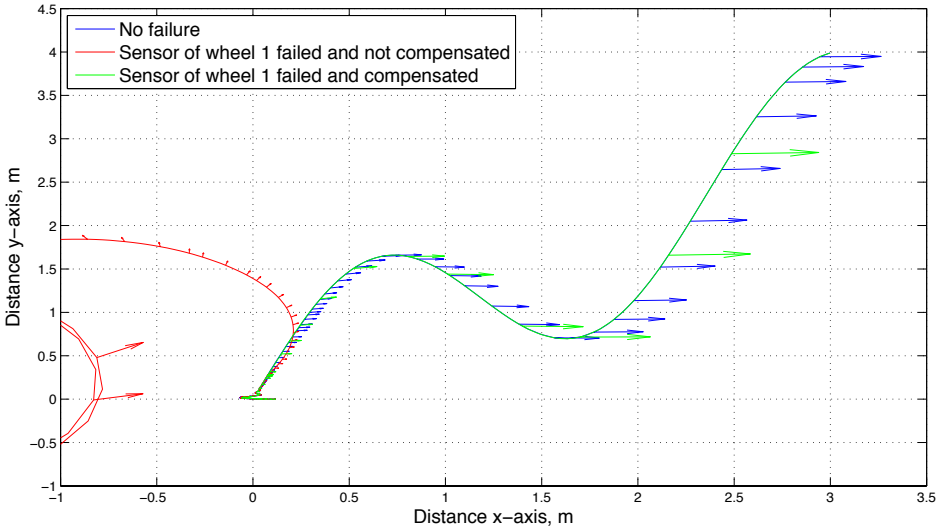


Figure 3.16: Projection of the tool tip on the x-y plane during trajectory tracking with tool-tip. The arrows represent the direction of the gripper.

Figure 3.17: The trace of the residual for 1st wheel-drive

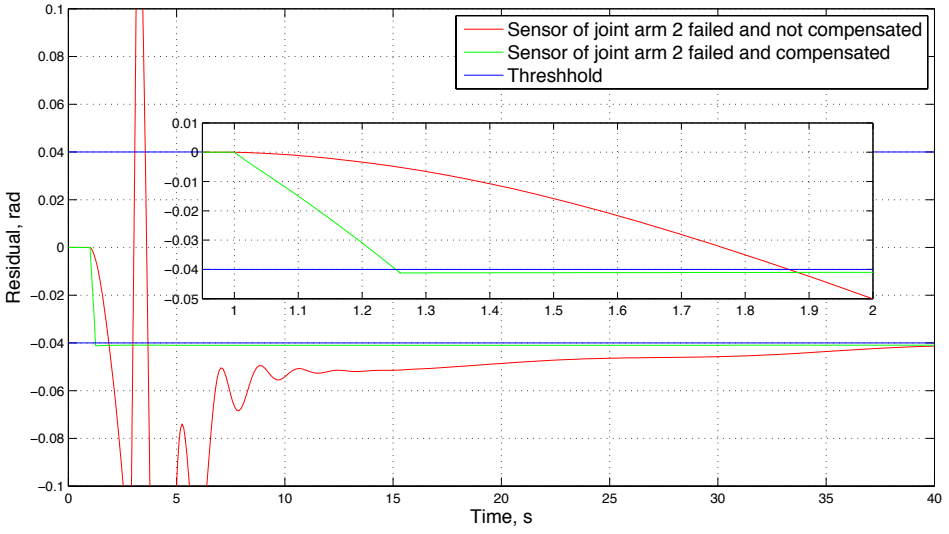


Figure 3.18: The trace of the residual for 2nd arm joint. Insert is the enlargement around first passing of threshold.

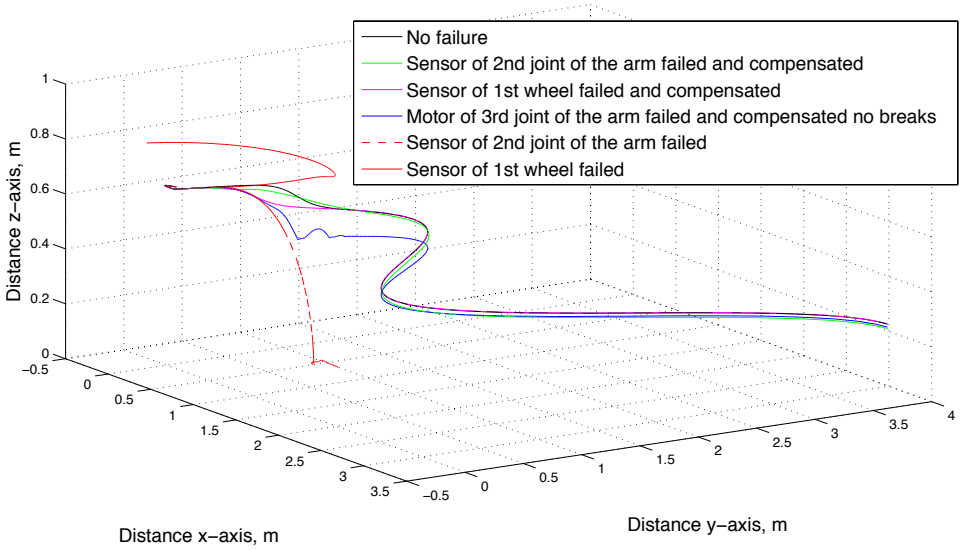


Figure 3.19: Distance between the tool-tip and the traced trajectory

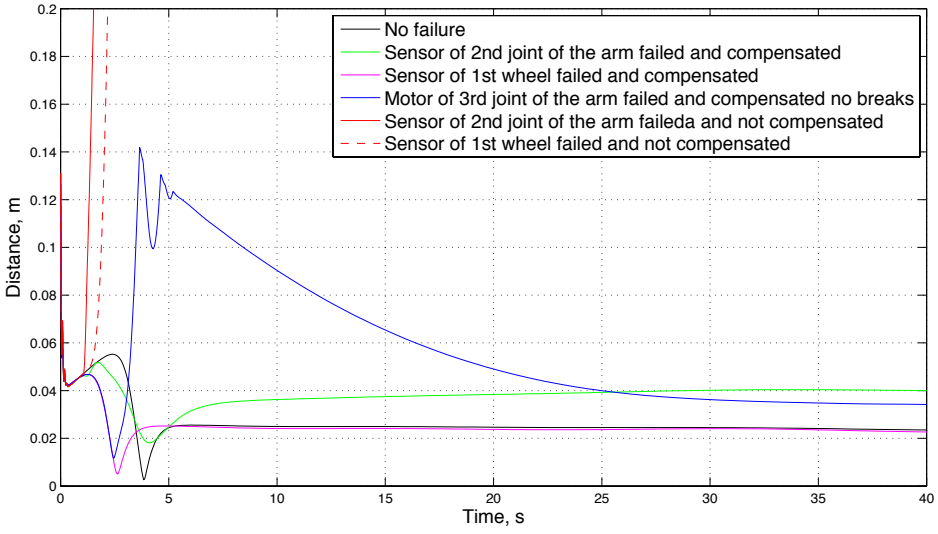


Figure 3.20: Trajectory of the tool-tip during tracking task

The trajectory of the tool tip is presented in Figure 3.19. The trajectory of youBot without fault-tolerant control (red color) is interrupted at the moment the tool-tip passes below the ground level.

In Figure 3.20, the tracking error is illustrated, which is represented by the distance between tool-tip and desired position on the trajectory, whereby orientation error is neglected in this graph. The nominal case without failure has a non-zero tracking error because of the simulated friction which is not compensated by the controller. It is seen in the figure that, in case of failure, the error grows rapidly if no compensation is present. With fault-tolerant control, however, the error converges to a small value similar to nominal case. The difference between nominal case and fault-tolerant control is due to the deviation in the sensor measurements introduced by the fault before it was detected. This deviation is equal to the threshold (Figure 3.18 green line follows closely to the blue after detection occurred). This deviation has less effect in the case of the wheel sensor failure than joint 2 sensor failure; because an additional rotation of a wheel for 0.04 rad (magnitude of the threshold) has less consequence on determining the position of the tool tip than a similar rotation of joint 2.

3.4.6 Discussion

Figures 3.17 and 3.18 can be used to judge the performance of the FDI system. The residuals respond rapidly to each type of failure, allowing to detect and isolate it. The residuals do not create false positive alarms.

Figures 3.15, 3.16, 3.19 and 3.20 can be used to judge the performance of the recovery procedure. Figures 3.20, 3.16 and 3.19 indicate that the proposed fault-tolerant control significantly improves the tracking performance with respect to non fault-tolerant behavior for failures in arm joints and wheel drives. The failure of the wheel-drive sensor is compensated completely, and the base moves almost in the same trajectory as the without failure (Figure 3.15). Fault-tolerant control maintains all required degrees of freedom for the tooltip such that both orientation and position of the tool tip can be tracked (Figure 3.16).

In case of joints without breaks, controller reconfiguration combined with powering off the broken joint can maintain part of the working envelope. It can be seen that the positioning error decreases at the end of the trajectory (Figure 3.20 blue line) when the set point moves closer to the ground and enters the remaining working envelope. The use of breaks in the joint allows to make the connection rigid and to recover desired performance faster as well as maintaining a bigger working envelope than without break.

3.5 Conclusions

A fault-tolerant control strategy that combines a compliance control for youBot as single kinematic chain, an observer based FDI and a recovery procedure was presented. Using fault injection, it has been shown that, in case of a sensor failure, a robot will react with un-controlled motion, which can be prevented using fault-tolerant control.

The presented FDI method uses a single Thao observer to generate residuals. This observer type allows for simple run-time computations as it does not require linearization and adjustments of the observer gains. The resulting residuals can be used to isolate failures of joints and wheel drives, which constitutes highest possible isolability for existing sensor arrangement using single observer architecture.

The chosen control strategy is shown to be able to compensate for failures. The resulted fault-tolerant control can compensate for failures of base wheel and joint drives, allowing to continue the trajectory tracking task. It has been shown that fault-tolerant control allows to increase the level of the fault protection. The robot without any fail resilience capability becomes fail safe, while a robot with fail safe actuator can be used as fail-operational system.

The recovery is done on the control-level; thus trajectory planning is unaffected and can be used to maintain performance of higher level tasks.

The working envelope of the robot is reduced in case of an actuation failure without engaging breaks on the broken joint as demonstrated in Figure 3.19. The fault-tolerant control as described in this chapter combined with fail-safe joints allows to reduce deviation from the desired trajectory during joint failures.

Experimental validation of the proposed method is the recommended future work. As the youBot does not have breaks installed on the joints of the arm, it is not possible to validate the efficiency of the proposed methodology without modification of hardware with respect to failures of the arm joints actuators. However, validation of the FDI method and the recovery procedure for the failures of base wheel actuators is possible. The validation can be performed in the same way as the verification was performed by applying the fault injection technique.

4

Ensuring Passivity in Distributed Architectures

A modular approach for building control systems opens up many applications, such as cooperative robots and load sharing for computationally intensive tasks. Furthermore, the ability to split controllers into multiple smaller ones enables application of CBSD re-use principles. One of the major barriers in building control systems in distributed way from a composition of components is faults in a communication medium such as latency, varying time delays and possible message loss.

In this chapter, we discuss a novel paradigm for controlling robots that require access to distributed resources and interact with an unknown environment. This paradigm relies on concepts of time domain passivity to track the exchange of energy between elements of the control system and the robots. We argue that keeping track of the energy exchange and appropriately reacting on energy inconsistent behaviors allow to make control strategies more robust and safe.

The passivity layer, discussed here, ensures that communication faults will not destabilize the system. The passivity layer can be used as an addition to existing control strategies to improve robustness as it works independently of used control law, similarly to a safety layer. The communication pattern based on the passivity layer does not put additional constraints on the framework or middleware used in the software implementation phase and therefore it can be implemented on top of existing ones. A case study demonstrates an application of the pattern and improvement in robustness with respect to time delays, on a simplified version of an object manipulation setup.

4.1 Introduction

The growing trend in robotics towards applications in an unstructured environment offers new challenges in control. Assisting, interacting and serving humans, new robots will literally touch people and their lives. While network communication widens the range of possible applications for robots, it brings a considerable challenge to control interaction with coordination of cooperation over a network, due to communication imperfections.

Tasks, such as complex assemblies, handling high payloads or manipulating flexible objects require the use of several manipulators (Siciliano and Khatib, 2008, p. 701).

Cooperative operations between multiple mobile manipulators offer valuable applications in the water (Stilwell and Bishop, 2000), in the air (Mellinger et al., 2013) and on the ground (Huntsberger et al., 2004).

A reliable way of building distributed control out of compose-able components enables a Component Based Software Development (CBSD) (Brugali and Scandurra, 2009) approach for controller development, thus decreasing development time for complex control algorithms. CBSD is a proven software engineering practice promoting re-use and thereby speeds up creation of a complex system considerably as discussed in Chapters 1 and 2. However, in all of component based software frameworks, the robot control algorithms are (normally) encapsulated in a single component. This limits the progress in creating reusable control functionalities.

In this chapter, a cooperative manipulation of a shared object is used as an example to present a control paradigm. A safe and stable manipulation of a shared object is not only a necessary step towards cooperative operations for robots, but it is also an example of a system of distributed components that combines a complex interaction in the physical domain and digital domain.

Manipulation of an object involves simultaneous control of clamping/internal forces and the dynamics of the object (Bonitz and Hsia, 1996). Many of the strategies proposed for grasping and manipulation of an object are based on what is known as passive physical-equivalent controllers (Khatib et al., 1996; Wimbock et al., 2011; Fassih et al., 2010; Stramigioli, 1998). These controllers provide solutions robust to uncertainties in environment, but they require complete information on the system states (Bonitz and Hsia, 1996; Stramigioli, 1999; Wimbock et al., 2006). This is a serious obstacle for implementing these techniques on distributed/networked systems. This is the case as, consistent real-time availability of states on the various nodes of the network cannot be easily implemented due to communication imperfections, such as time delays and message loss. As a consequence of these inconsistencies, the system performance degrades and instability can occur (Bemporad et al., 2010).

Various methodologies have been proposed to maintain stability of a system in the presence of communication imperfections. For example, for linear systems a compensator can be defined to have a specific margin of robustness with respect to time delays and message loss (Bemporad et al., 2010). In nonlinear tele-operation systems, a control based on the passivity concepts is used to effectively treat communication imperfections (Niemeyer, 2004). Energy-based concepts offer an effective approach to analyze system performance. From an energy-based point of view, discretization of a continuous time process and time delayed communication introduce discrepancies in the system energetic behavior, which could be resolved for example by using variable dissipation (Ryu et al., 2004; Franken et al., 2009). Similarly, these concepts can be extrapolated to autonomous control, where communication of several robots is required to obtain coordinated control.

As it will be shown later, enforcing proper constraints on the energy exchange between subsystems provides a way to ensure passivity of the overall system. To impose and handle such energy constraints for a controller, the passivity layer (PL) is devised,

based on the algorithms published in Franken and Stramigioli (2011). What is called the passivity layer (PL) is a combination of algorithms that computes precisely the energy flow between the discrete-time and continues-time parts of the system, tracks the energy exchange between the components of the control system and adjusts control signals to comply with the energy constraints. The PL acts similarly to the safety layer (Figure 1.3). A safety layer remains dormant during normal operation and takes over control in case of undesired behavior of the controlled system. The PL works independently from the control law and can be added into the system as a part of safety.

In this chapter, we present a framework suitable for the control of robots that interact with an unknown environment and require access to distributed resources, e.g. computation, actuators or sensors. The improvement in robustness of the distributed control is demonstrated in an experiment that mimics cooperation of two robots in a manipulation of a single shared object. The method is described in Section 4.2, providing details of the idea of energy-based constraints, the algorithms of PL and architecture of a single controller-component and a composition controller-components. The mathematical model of the experimental setup and the architecture of the controller are presented in Section 4.4. The mathematical model is used to demonstrated the effect of communication imperfections on controller performance. The effects of the introduction of the PL to the experimental setup are demonstrated in Section 4.5. The analysis of the results is done in section 4.6. The conclusion are presented in section 4.7.

4.2 Passivity Layer

4.2.1 Passive systems and energy-based constraints

Passivity is a property of a dynamic system that can be used to prove stability of the system. The theoretical concepts and properties of passive systems are, for example, illustrated by van der Schaft (2000). In systems theory, the interconnection of two systems can be modeled by a bi-directional information (signals) exchange, which is termed a *port*. The energy exchange between these two systems will be a function of these signals in the port. In passivity theory, a system is considered passive if the energy that can be extracted from all its ports is finite and bounded by its initial energy.

In applications of system theory to robot control, actuators can be modeled by a port, through which energy can be supplied to/extracted from the robot. This port is mathematically represented by the pair (F, v), which denotes respectively the force generated by the actuator and its collocated velocity. The product $F \cdot v$ represents the power supplied by the actuator to the robot. The energy supplied form time 0 to time t can therefore be expressed as:

$$H(t) = \int_0^t F(s)\, v(s)\, ds \tag{4.1}$$

where $F(s)$ is the actuator force and $v(s) = \dot{q}(s)$ is its velocity.

The definition of a passive system states that the energy extracted from the system

should not be greater then its initial energy (Willems, 1972b,a). This is a constraint, which, in case of a controller for a single actuator with a defined energy storage ($H(0)$), can be expressed as:

$$H(0) \geq -\int_0^t F(s)\dot{q}(s)ds \tag{4.2}$$

Enforcement of the energy constraint (Eq. 4.2) ensures passivity of the system. The core idea of the proposed approach is to satisfy this energy constraint by influencing the control signal. Combining information about allowed energy exchange and a measured feedback signal, the energy constraint is transformed into a control signal constraint.

Consider a typical actuation pattern for a robotic system in which the controller has an impedance causality, *i.e.* the force is the generated control signal and the corresponding motion is measured. Moreover, assume that the control signal is implemented as zero order hold (constant value between two sample-times), which is also typical for the digital robot control. Using these assumptions, the energy (ΔH_k) obtained/lost by the controller during one sample interval can be expressed as:

$$\begin{aligned} \Delta H_k = -\int_{T_{k-1}}^{T_k} F(s)\dot{q}(s)ds = -\int_{T_{k-1}}^{T_k} F_{k-1,k}\dot{q}(s)ds = \\ -F_{k-1,k}\int_{T_{k-1}}^{T_k} \dot{q}(s)d = -F_{k-1,k}(q_k - q_{k-1}) \end{aligned} \tag{4.3}$$

where $T = T_k - T_{k-1}$ is the sampling interval, $F_{k-1,k}$ is the force applied during the sampling interval and q_k, q_{k-1} are respectively the positions of the actuator at the start and the end of the sampling interval.

Combining Eq. 4.2 and Eq. 4.3 the energy constraint can be computed precisely as follows:

$$H(0) > -\sum_{j=0}^{k} F_{j-1,j}(q_j - q_{j-1}) \tag{4.4}$$

Eq. 4.4 can be used in two ways to identify if the system is behaving passively and to plan the control signal such that passivity is ensured.

Overall, for each action, a robot has an associated energetic cost and this cost can be explicitly represented. To maintain passivity according to Eq. 4.4, the movement of and forces created by the robot have to be restricted. Ensuring passivity of the controller leads safe and stable operation.

4.2.2 Algorithms

To ensure safe and stable operation of the controller, using proposed based energy constraint (Eq. 4.4) a PL is employed. The PL plays the role of a safety layer (Figure 1.3) using one step ahead prediction to satisfy the energy constraint (Eq. 4.4). A PL

is employed to perform a final check of energy-based constraints before applying a control signal. The PL consists of four parts (Figure 4.1).

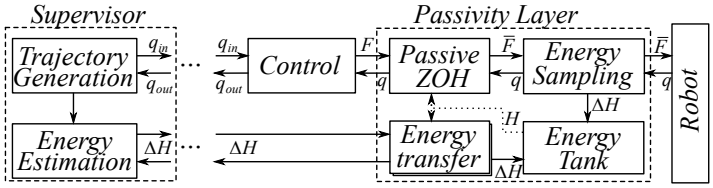


Figure 4.1: Passivity layer structure, data flow diagram

Energy sampling

The function of the Energy Sampling block is to measure the energy exchange between the discrete-time controller and the robot. As noted in the previous section (eq. 4.3), the energy exchange within one sample period can be computed as:

$$\Delta H_k = -\bar{F}_{k-1}(q_k - q_{k-1}) \tag{4.5}$$

where $\bar{F}_{k-1}$ is the force applied to the robot. Note that this force might differ from the force commanded by the controller (Figure 4.1) as discussed in the Passive ZOH subsection. The energy quanta ΔH_k, the exchanged energy over one sampling interval, is recorded by a lossless Energy Tank.

Energy Tank

The lossless Energy Tank is introduced to keep track of the energy state. The Energy Tank provides information about available energy to the other blocks, such that energy spending can be controlled (dashed line to Passive ZOH and Energy Transfer (Figure 4.1).

Note that in Figure 4.1 the exchange of energy quanta (ΔH) between the Energy Tank and connected blocks is marked differently (solid and doted lines). This is done to emphasise that the exchanged energy quanta (solid lines) added or subtracted from the amount of energy available for the controller, while providing information about a current energy state (doted lines) does not affect the energy state. The connected blocks must take care to use only available energy, as the amount of energy available for the controller is a conserved quantity.

The level of the energy tank should be interpreted as a tight energy budget which is used to pay for all actions. The process of restricting the energy consumption is described in the subsections named Passive ZOH and Energy Transfer.

The sum of used and gained energy quanta constitute the change in the energy state for the current sample time:

$$H_k = H_{k-1} + \Delta H_k^{ES} + \Delta H_k^{ET} \tag{4.6}$$

where ES stands for Energy Sampling and ET stands for Energy Transfer.

Passive ZOH

The goal of the Passive ZOH block is to enforce the energy-based constraints on the level of the control signal. To up hold the passivity property, the control signal has to be energetically consistent, violation of the energy constraints is an indication of non-passive behavior. If the control signal is energetically inconsistent, e.g. violates the energy constraints, it has to be modified.

The energy constraint (Eq. 4.4) has to be examined to identify an energy inconsistent signal. This energy constraint (Eq. 4.4) for sample time $k+1$ can be expressed as:

$$H_k > -\Delta H_{k+1} = F_k(q_{k+1} - q_k) \tag{4.7}$$

where F_k is the control signal commanded at sample time k and maintained constant until the sample time $k+1$; q_{k+1} is the measured position at the sample time $k+1$; H_k is the energy state (computed in Energy Tank) at sample time k of the controller and ΔH_{k+1} stands for energy quanta obtained/lost by the controller during sample interval $k, k+1$. Taking into account that $\Delta H_{k+1} > 0$ leads to increase of the energy state of the controller H_k(from Eq. 4.6) and Eq. 4.6, three types of control signals can be identified:

1. F_k results in $\Delta H_{k+1} \geq 0$. Such signal is always leads to extraction of energy from the controlled system, therefore it is also energy consistent independently of the energy state of the controller.

2. F_k results in $0 > \Delta H_{k+1} > -H_k$. Such signal is also energy consistent, however, for such signal to exist $H_k > 0$.

3. F_k results in $-H_k > \Delta H_{k+1}$. Such signal is energy inconsistent execution of this signal will lead to loss of passivity.

Saturation of the control signal can be used to limit the energy exchange. This approach has been shown to produce maximum transparency for control while enforcing a passive behavior (Franken and Stramigioli, 2011), and therefore used here.

The concept of Passive ZOH is as follows:

- if actions require less energy than available in the energy tank (the signal types 1 and 2 above), the control signal is applied without any modification.

- if actions require more energy than available (the signal type 3 above), the control signal is reduced in a magnitude to prevent a violation of the energy-based constraints.

The modification of the control signal is made based on the limiting/saturation functions. The fundamental limit which has to be imposed to maintain passivity is a zero level of the energy tank ($H_k < 0$). This limit can be enforced as follows:

$$\bar{F}_k = \begin{cases} F_k & H_k > 0 \\ F^* & H_k \leq 0 \end{cases} \tag{4.8}$$

where F_k is the signal computed by the control (Figure 4.1), $\bar{F}_k$ is the control signal applied to the robot, F^* is the control signal that satisfies the $\Delta H_{k+1} \geq 0$ condition.

F^* depends on the dynamics of the controlled system. The model of system dynamics can be used to design F^* as a function of measured states. An example of F^* is a damping injection as presented in Franken and Stramigioli (2011). In case of no knowledge about system dynamics, this signal can be chosen as $F^* = 0$, which would be based on Eq. 4.7 results in $\Delta H_{k+1} = 0$. In other words, the control signal is reduced to zero to prevent further energy exchange.

To prevent the type 3 signals, that may result in a "temporary loss of passivity" ($H_k < 0$), the energy exchange for the next sample can be predicted by Eq. 4.7and it can be used to define an additional energy constraint. Assuming that all available energy can be consumed in the following sample ($H_k = \Delta H_{k+1}$), the control signal upper and lower bounds can be expressed as:

$$F_k < \frac{H_k}{q_{k+1} - q_k}, \text{for } q_{k+1} - q_k > 0 \tag{4.9}$$
$$F_k > \frac{H_k}{q_{k+1} - q_k}, \text{for } q_{k+1} - q_k < 0$$

Eq. 4.9 require estimation of q_{k+1} based on the force to be applied F_k. Obtaining precise estimate is not possible in a real life system, however these equations define the range of allowed signals, and therefore it is possible to use what can be called a pessimistic estimate of q_{k+1}. We define a pessimistic estimate of q_{k+1} as any estimate that results in range of signals defined by Eq. 4.8 and Eq. 4.9 smaller then a real value of q_{k+1}.

The concept of pessimistic estimates can be easily explained through an example. Assume that F_k results in $q_{k+1} \in (q^l_{k+1}, q^h_{k+1})$ then

$$q^h_{k+1} - q_k > q_{k+1} - q_k > q^l_{k+1} - q_k \tag{4.10}$$

Eq. 4.9 is computed as

$$F_k < \frac{H_k}{q^h_{k+1} - q_k} < \frac{H_k}{q_{k+1} - q_k}, \text{for } q^h_{k+1} - q_k > 0 \tag{4.11}$$
$$F_k > \frac{H_k}{q^l_{k+1} - q_k} > \frac{H_k}{q_{k+1} - q_k}, \text{for } q^l_{k+1} - q_k < 0$$

Since, the pessimistic estimates of q_{k+1} result in tighter bound on the control signal, the constraint Eq. 4.7 is satisfied. Obtaining pessimistic estimates is relatively simple for any system by including confidence intervals in the estimate. It is interesting to note, for confidence intervals such that $q^h_{k+1} = +\infty$ and $q^l_{k+1} = -\infty$, the allowed control signal is $F_k \to 0$, that is preventing further energy exchange.

Eq. 4.11, despite of including the confidence interval in estimation of the q_{k+1}, is still a hypothesis, and there is a probability that the actual value will lie outside of the confidence interval. This would result in a "temporary loss of passivity" ($H_k < 0$), thus Eq. 4.8 has to be included in the algorithm to recover from such situations.

Combining Eq. 4.8, 4.9, 4.11 and the idea of saturation, the control signal applied to the robot can be determined as:

$$\bar{F}_k = \begin{cases} F_k & \frac{H_k}{q^l_{k+1}-q_k} < F_k < \frac{H_k}{q^h_{k+1}-q_k}, H_k > 0 \\ \frac{H_k}{q^l_{k+1}-q_k} & F_k < \frac{H_k}{q^l_{k+1}-q_k}, H_k > 0 \\ \frac{H_k}{q^h_{k+1}-q_k} & F_k > \frac{H_k}{q^h_{k+1}-q_k}, H_k > 0 \\ F^* & H_k \leq 0 \end{cases} \tag{4.12}$$

Energy transfer

The Energy transfer block is responsible for the communication of energy between the distributed and communicating controllers. The type of algorithm for the Energy transfer block will depend on the application. In the simplest case, the exchange is performed to equalize the levels of the energy tanks in a set of communicating controllers, for example, using the simple energy transfer protocol defined in (Franken and Stramigioli, 2011). The sum of the energy levels in the system should not be affected by the communication process.

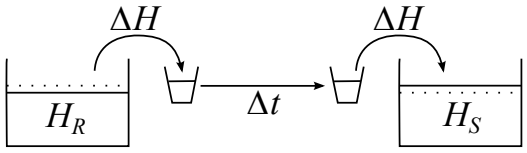


Figure 4.2: Intercomponent energy communication

Within the PL, the energy exchange between communicating controllers is organized in the form of energy packets. Whenever an energy packet (ΔH_k) has to be sent from one controller to another, it is first subtracted from the sender's energy tank ($H_{S,k}$) and then sent through the communication channel. On arrival, an energy packet is added to the energy tank ($H_{R,k+n}$) of the receiver (Figure 4.2). In this way, the energy on the receiver side is delayed by the communication time (n sample-times), but overall energy in the system remains constant:

$$\begin{aligned} H_{S,k+1} &= H_{S,k} - \Delta H_k \\ H_{R,k+n+1} &= H_{R,k+n} + \Delta H_k \end{aligned} \tag{4.13}$$

Energy Estimate

To perform trajectory tracking, the energy used by the controlled system has to be estimated. Energy estimation depends on the controlled system, task and control quality measures. The energy estimation block (Figure 4.3) essentially performs a simulation of the controlled system dynamics moving along the trajectory with desired control and provides an estimation of energy consumed by the system.

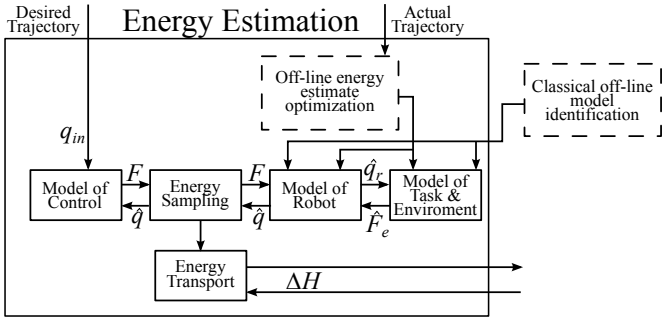


Figure 4.3: A complete structure of Energy Estimation block from Figure 4.1

4.3 Application of PL to distributed control

4.3.1 System architecture

In order to use PL within a distributed control system, a system architecture should be defined. Application of PL to distributed control systems imposes rules on the architecture.

Interconnection of components is required for correct operation of PL in distributed control, where interconnection is defined as bi-directional exchange of energy. Interconnection of the components is achieved by bi-directional connection of energy exchange ports of PL (left side of Energy Transfer in Figure 4.1). To maintain energy consistency throughout the system, interconnection has to be defined between precisely two energy exchange ports. In other words, an energy exchange port can only be connected to only one other exchange port (one-to-one connection). Other types of connections (such one-to-many) will jeopardize energy content in the system. In case a component has several peer components, it has to have an adequate number of energy exchange ports; thus such component will be explicitly controlling energy distribution between its peer-components. In Figure 4.4, the interconnection of the components is indicated using half-arrow of bond-graph notation, to emphasise the similarity between energy-consistent interconnections between control-components and energy-consistent interconnection in physical systems.

If the control system that features a PL is designed to perform a trajectory tracking, the component that generates trajectory will also receive an additional function of energy estimation. The estimate is made on the basis of the models of the robot, the environment and the control law. The estimate of the required energy is, essentially, a part of a trajectory. In Figure 4.4, the combination of energy estimator and trajectory generator is named the Supervisor (similarly to Intrinsically Passive Control concept (Stramigioli, 2001)). The energy propagates simultaneously with the trajectory set-points, through the composition of components based on the energy transfer protocol. Since PL limits a control signal based on the energy state of a component, the

energy received together with the trajectory set point "enables" the control signal prescribed by the trajectory.

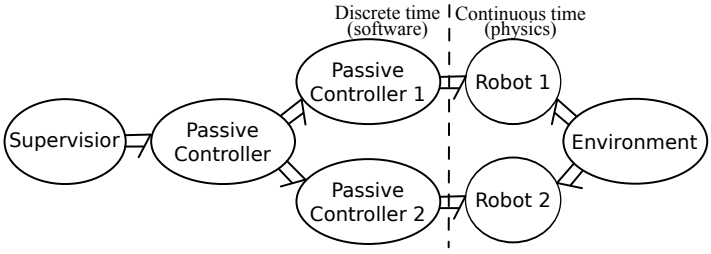


Figure 4.4: Structure of connections of supervisory control, distributed passive controllers and robot. Note that port connection use half-arrow of bond-graph notation to emphasise energy-consistent interconnection

The control architecture can be represented as a graph that covers a complete robot or set of robots that work together. The structure of the graph and the input from the supervisor shapes the system behaviour. An example presented in Figure 4.4 indicates major elements of the control architecture.

The Passive Controllers 1 and 2 (Figure 4.4) provide a control of the actuators, which is passive in nature and thus guarantees stability for interaction with an environment. Optionally, the component composition can contain other blocks (Passive Controller in Figure 4.4) that provide additional behaviour or distribute the control objectives and route the energy flow. These blocks do not have direct connections to the actuators, but have energy-consistent interconnections with other controllers. Such arrangements allow to incorporate sensory information or work with multiple set of actuators without modifying the control laws for the actuators.

Due to use of PL, inconsistencies caused by communication failures will not lead to unstable behavior, but to reduction of performance. The modeling errors in the energy estimate as well as trajectory generation would also lead to reduction in performance, as noted in Section 4.2.1. Note that reduction in performance is also detected by PL as not enough energy available in the Energy Tank. This information can be propagated to the Supervisor (Figure 4.4) to be used in planning of corrective actions.

An important consequence of using PL in distributed control is so called local temporal autonomy (LTA) (Bemporad et al., 2010). LTA means that a set of components once being disconnected from the main part of the control system will continue to execute set task in safe, and stable manner. In Figure 4.5, a connection loss between control components is shown. Since passivity of each set of component is ensured by the PL, disconnected part of the control system continues to interact with the environment in a stable manner that was specified before communication was lost. After bi-directional communication is reestablished, a part of the control system, which was disconnected, will compensate for any energy disbalance using initially defined Energy Transfer protocol.

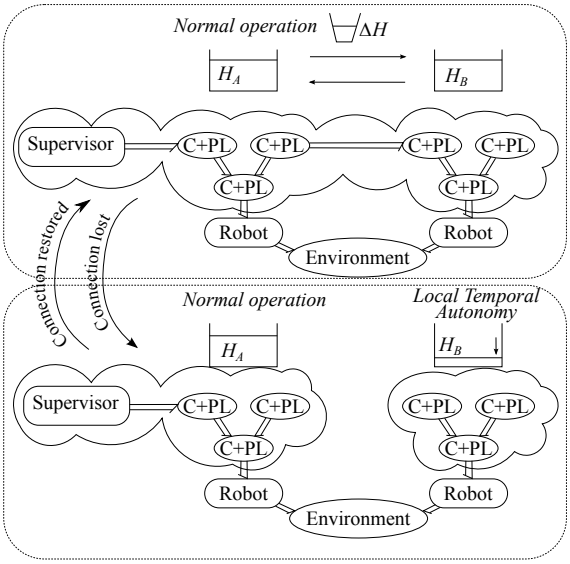


Figure 4.5: Energy levels during connection loss, "C+PL" is a passive controller-component that features the passivity layer

4.3.2 Component architecture

The introduction of the PL into a component structure results in two types of information flows:

- **DT-CT** → Communication to actuators covers the process of interconnecting digital computation to a continuous dynamic process. Passivity of this connection between discrete time (DT) domain and continuous time (CT) domain is enforced by the PL using the following algorithms: Energy Sampling (sec. 4.2.2) and Passive Zero-Order Hold (sec. 4.2.2). The signal generated by this type of communication transports energy from the DT domain to the CT domain and vice versa. This signal must be protected against communication imperfections. In other words, this signal has to be directly passed to the controlled actuator.

- **DT-DT** → Communication between components covers the process of data exchange between two discrete-time (DT) systems,*i.e.* two control components. The communicated data (q_{in}, q_{out} depicted in Figure 4.1) between the controllers is not affected by the PL. The energy exchange is explicitly separated from it and governed by the energy transfer block of the PL. Due to this explicit separation of transferred energy and communicated data, faults and imperfections can be tolerated in this type of communication.

Figure 4.6 depicts the information transfer between various software layers. The device layer consists of the hardware: actuators, sensors, communication lines etc. The

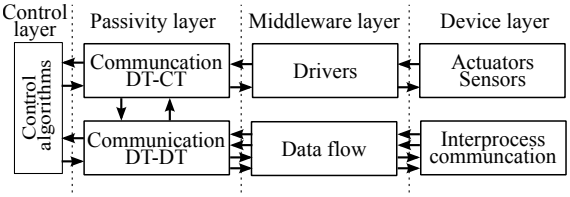


Figure 4.6: Signal exchange in a single component including the PL. Note double arrows in the bottom section that indicate separate transfer of the energy and controller set points

middleware layer provides an abstraction from hardware by handling access to the hardware and communication between components as described above. The control layer contains the implementation of the actual control laws, which are designed to provide the desired performance. The passivity layer handles communication components and enforces the passive behaviour of the control laws.

4.4 Case study

To demonstrate the use of the PL during the cooperation of two robots, a simple use case is selected. As noted in the introduction, coordinating manipulation over a network can lead to an unstable behavior due to network induced-imperfections, such as time varying delays and packet losses. This section presents the model of the use case system and conditions of instability during object manipulation.

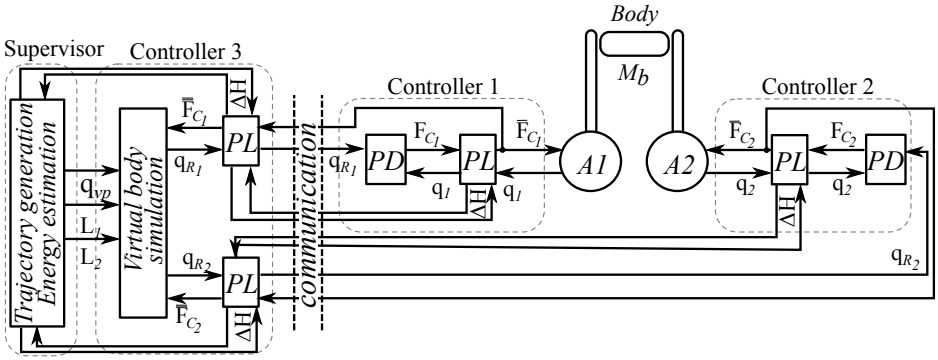


Figure 4.7: Cooperative manipulation with communication

The object-level control based on the virtual springs definition is a versatile approach to object manipulation (Wimbock et al., 2011; Stramigioli, 1999; Khatib et al., 1996), used to achieve robust control for the internal forces and the object dynamics. In this approach, virtual springs definitions with various topologies are used to define the internal forces and the object dynamics. In this case study, the object manipulation strategy proposed in (Stramigioli, 1999) is assumed, as it features two major elements

in the control structure: the virtual object and the springs definition. Implementation of this strategy on a distributed/networked system in the worst case scenario is separation between all control elements. In other words, the control of each robot and the simulation of the virtual object will become separate control components which have to communicate over the network.

The simplest way to reproduce this situation is by using two one-dimensional actuators (A1, A2) to grasp a rigid body (Figure 4.7). These actuators are used to impose internal forces on the body and to control the position of the body. Communication is used to synchronize the states and the desired set points.

4.4.1 System model

A simplified dynamic model of the actuator consists of a mass, M_i, and friction, B_i. The position, $q(t)$, and the velocity, $\dot{q}(t)$, can be derived as follows:

$$\begin{aligned} F_{B_1}(t) + F_{C_1}(t) &= M_1 \ddot{q}_1(t) + B_1 \dot{q}_1(t), \\ F_{B_2}(t) + F_{C_2}(t) &= M_2 \ddot{q}_2(t) + B_2 \dot{q}_2(t) \end{aligned} \tag{4.14}$$

where the subscripts indicate the number of the actuator, $F_{B_i}(t)$ is the reaction force of the body due to compression during grasping and $F_{C_i}(t)$ are forces commanded by the controllers. The controller is a PD-controller and formulated as a spring-damper combination:

$$F_{C_i}(t) = K_{P_i}(q_{R_i}(t) - q_i(t)) - K_{D_i}\dot{q}_i(t), \quad i \in \{1,2\} \tag{4.15}$$

where K_{P_i}, K_{D_i} are the proportional and derivative gains of controllers for each actuator, and $q_{R_i}(t)$ are the set points, generated by the virtual body simulation. The virtual body simulation is executed at the supervisor component (Figure 4.7) and defined as:

$$\begin{aligned} M_v \ddot{q}_v(t) + B_v \dot{q}_v(t) + K_v(q_v(t) - q_{vp}(t)) &= \\ F_{C_1}(t) + F_{C_2}(t) & \\ q_{R_i}(t) = q_v(t) + L_{R_i}(t), \quad i \in \{1,2\} & \end{aligned} \tag{4.16}$$

where M_v is the mass of the virtual body, B_v and K_v are damping and stiffness, defined for the virtual body and will be reflected on the real manipulated object; q_{vp} is a desired position of the virtual body. $L_{R_i}(t)$ are the rest lengths of the virtual springs that define the internal forces.

The object is modeled as a mass, M_b, friction, B_b, and contact stiffness K_s. It is modeled as a linear system, the contact discontinuities are neglected in the stability analysis in Section 4.4.2 for simplification.

$$\begin{aligned} M_b \ddot{q}_b(t) + B_b \dot{q}_b(t) &= F_{B_1} + F_{B_2} \\ F_{B_i} &= K_{S_i}(q_b(t) - q_i(t)),\ i \in \{1,2\} \end{aligned} \tag{4.17}$$

where $q_b(t)$ and $\dot{q}_b(t)$ indicate the position and the velocity of the bar.

The controllers and the supervisor communicate over a network. This network is implemented as periodic handling of messages, with the same period as the controller runs on, and without buffering of messages. Periodic handling of messages was chosen instead of processing messages on arrival as this simplifies the analysis performed in Section 4.4.2. Thus the model of the communication of a signal (u_k) is:

$$\hat{u}_k = u_{k-n} \tag{4.18}$$

where $\hat{u}_k$ is the signal available for the receiving part of the controller and n is the number of periods the signal (u_k) is delayed.

4.4.2 Stability analysis and robustness

To demonstrate the necessity of the PL in cooperative object manipulation, the stability robustness (Hu and Yan, 2007) of the system is analysed. The case study shows that the variation of parameter values of the manipulated object and of the communication can easily jeopardize the system.

To analyse stability of the system without PL, the communication is modeled explicitly. Following the methodology described in (Bemporad et al., 2010), the state space representation has been obtained. It is discretized and a model of the communication is included. The complete system including the communication model is presented as a linear system with a transition matrix, F, and the input matrix, G, (see derivation in Appendix C).

$$\xi_{k+1} = F\xi_k + Gu_k \tag{4.19}$$

where ξ_k is the extended state that includes delayed signals, u_k is the grasp control set points, that are the rest lengths of the springs and the position of the virtual body.

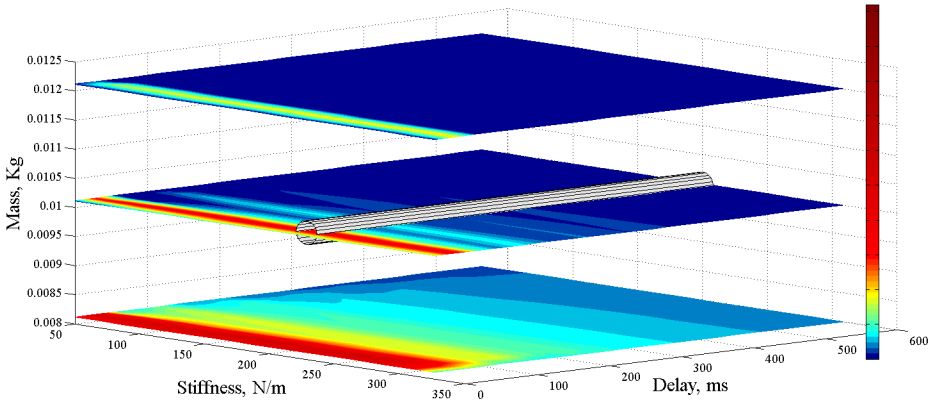


Figure 4.8: Stability region for cooperative object manipulation with various time delays, mass and stiffness of manipulated object. Brighter colours denote lower settling times, dark blue is unstable. The grey cylinder indicates the region of the *experiment type II* (Section 4.5)

Stability of the system can be concluded by examining the eigen values of the matrix F. Using the griding approach (Figure 4.8) the stability region of the system with respect to time delays, stiffness and the mass of the manipulated object can be obtained.

Figure 4.8 shows that the stability robustness with respect to time delays and object parameters is limited. Increase of the time delays will lead to decrease of the stability robustness with respect to variation of the parameters of the manipulated object: mass and stiffness (Figure 4.8). Thus additional measures are required to ensure stability of the system for all types of objects. Ensuring passivity of the control allows ensuring a stable interaction with a passive environment, consequently increasing robustness of the whole system as it is shown in the experiments.

4.4.3 Implementation of the energy estimation in the supervisor

The estimation of the required energy to perform a task is based on a simplified model of the supervised system, to demonstrate the robustness of the PL approach. Only two effects are taken into account, the dissipation by friction simulated on the virtual body and the required grasp energy. The virtual body moving along the trajectory dissipates energy due to the simulated friction. That energy has to be supplied by the supervisor to compensate irreversible losses. Furthermore, the required energy to grasp an object (grasp energy) is defined by the rest lengths of the virtual springs (L_{R_1}, L_{R_2}) and is estimated with a linear spring model. Therefore, the energy supply required by the supervised system is obtained as a sum of the above mentioned energy terms:

$$\delta H = \underbrace{\mu_f \dot{q}_{vp}^2}_{\text{simulated friction}} + \underbrace{\mu_g (\dot{L}_{R_1} L_{R_1} + \dot{L}_{R_2} L_{R_2})}_{\text{grasp energy}} \tag{4.20}$$

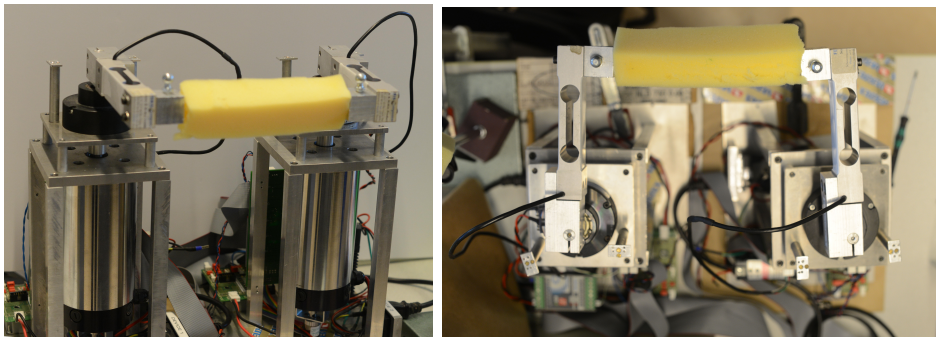
where $\mu_f = B_v$ is the dissipation coefficient on the virtual body and $\mu_g = K_{P_i}$ is the linear spring coefficient of the virtual springs.

The energy is added to the energy tank of the supervisor in energy packages sufficient to execute one step in the trajectory (Eq. 4.20) and then distributed to the controller components. Using this approach, the total energy in the system only sufficient to execute already commanded changes in the trajectory. This results in minimal local temporal autonomy of the control components thus inconsistencies in control signals induced by communication can be detected by PL on time.

4.5 Experiments

In this section, experiments to demonstrate the behavior of the system with and without the PL are described. The experiments are performed using the setup shown in Figure 4.9a. The setup consists of 2 one-DOF devices powered by direct-drive DC motors. Both devices are controlled from a single embedded controller running on a real time Linux computer. A symmetric constant delay is implemented in the communication channel between each two controller and the supervisor. Note that the amount of time delay stated below is delay of the message in one direction, therefore, round trip delay between the components is double of the stated amount. Two types of ex-

periments are presented here, with constant (Experiment type I) and with increasing time delay (Experiment type II).



(a) Side view **(b)** Top view

Figure 4.9: Experimental setup, the diagram system architecture is presented in Figure 4.7

Experiment type I: Time delays are kept constant (10 ms and 300 ms) while the object is grasped and a disturbance is applied to the object.

Experiment type II: The object is grasped and a reference trajectory (smoothed square wave) has to be tracked by the object. Time delays in the system are increased starting from 10 ms with a step of 50 ms every 5 seconds until the object is dropped. At every increase of time delay, 5 consecutive messages are lost.

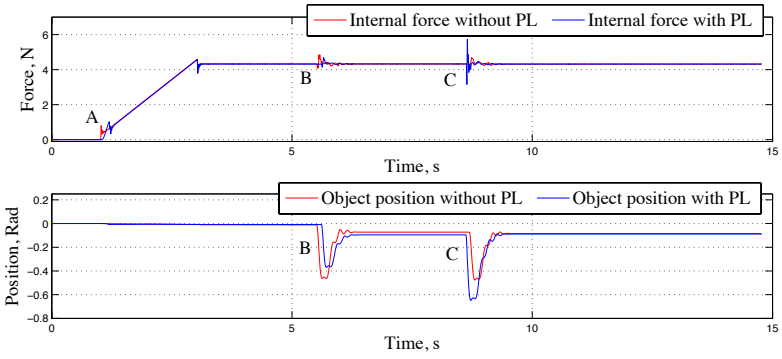


Figure 4.10: *Experiments type I:* Time delay of 10 ms

4.6 Results and Discussion

The behavior of the system while running experiments type I is shown in Figure 4.10 and Figure 4.11. Grasping the object with small time delays (Figure 4.10) provides a base line performance of the system. In both plots several spikes are present. Spikes

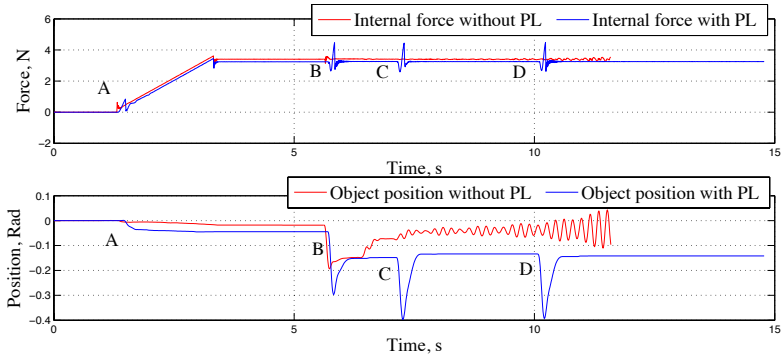


Figure 4.11: *Experiments type I:* Time delay of 300 ms

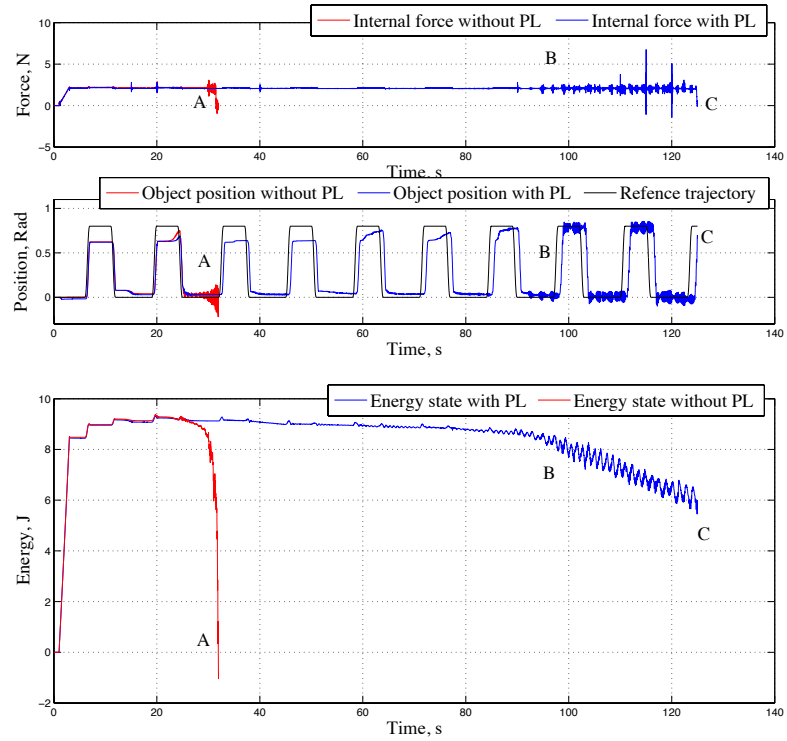


Figure 4.12: *Experiments type II:* The system behavior during grasping with a trajectory tracking. The time delays are increasing every 5 seconds by 50 ms. The experiments is stopped when the grasped object is lost.

in the beginning (A in Figure 4.10) attributed to the contact with a hand-held object. The spikes (B, C in Figure 4.10) in the trace of object position are the disturbances created by applying a force to the object and then releasing it. In both plots we can see that the behavior of the system with and without PL is similar. After each perturbation the system returns to the reference position (0). A static error with respect to reference position (0) is due to non-linear friction of the actuators, which is not compensated in the adopted control strategy.

In Figure 4.11 the behavior of the system is presented in case of 300 ms time delay. Without PL, when disturbance was induced to the system (B in Figure 4.11), the object did not return to its previous position, instead the vibrations grow in amplitude and the system becomes unstable. With PL the system recovers after the disturbance (B, C, D in Figure 4.11), however the object did not return to the original position, i.e. the static error has increased. This behavior is expected as the energy constraints concept of the PL restrict movements of the system.

The computation of the energy exchange provides a detection mechanism for unstable behavior. Figure 4.12 depicts the evolution of the energy state, as the system performs a trajectory tracking. Increase in time delays results in vibrations of the object (Figure 4.12 object position). These vibrations can be detected as the energy dissipates (Figure 4.12 energy state without PL). The energy state of the system without PL drops below the energy of the grasp and becomes negative at the moment when the object is dropped. This change in the energy state corresponds to a non-passive behavior and can be detected by the PL.

The PL enforces passive behavior by controlling the energy exchange. The reduction in the energy content triggers the PL to reduce energy flow thus preventing further destabilization of the system and further loss of energy. This mechanism can be perceived not only as protection from instability, but also as a negative feedback loop on the energy content of the system. An overestimation of the energy level combined with controller inconsistencies results in vibrations, which dissipate energy until a correct estimation level is reached (energy of the system decays as time delay grows and inconsistencies in the system states accumulate (B in Figure 4.12) between 40-100 seconds). An acute overestimation will, however, result in the prolonged vibrations before PL will be able to detect instability. This can be seen at the end of the experiment (C in Figure 4.12). The discrepancies in the estimation of the energy, required to preform tracking, and the inconsistencies introduced into the control by time delays accumulate as time delay increase and result in vibrations build up. These vibrations lead, eventually, to loss of the object. An underestimation of required energy results in increased static errors. The PL prevents the controller from performing the energy exchange, which effectively means that movements of the robot are restricted. This can be seen in Figure 4.11 the object has not returned to the original position after the disturbance was removed. If the energy use is correctly estimated, the PL has no effect on the system performance as reflected in Figure 4.10.

The increase of stability robustness should have a minimal effect on system performance during the normal operation. However, as the system diverges from the oper-

ating conditions, the performance of the system is allowed to degrade gradually, similarly to fault tolerance algorithms (Visinsky et al., 1994). Continuing the analogy with the fault tolerance algorithms, the PL can be interpreted to have a double role: detection of unstable behaviors and enforcing passive ones.

4.7 Conclusions

In this chapter we presented the basis for a methodology which allows to implement physically based controllers in a distributed framework by ensuring a passive behavior. It combines ideas of Physically Based Control (Hogan, 1985), Intrinsically Passive Control (Stramigioli, 2001) and time domain passivity (Franken and Stramigioli, 2011) into an architectural pattern named Passivity Layer (PL). The PL approach allows to increase robustness of such controllers without affecting its initial performance. This was demonstrated using an example of distributed control to implement grasping. It has been shown that with PL stable grasp can be maintained with a communication delay of 1.2 s.

The components of the distributed control are designed to have a certain level of temporal local autonomy that allows to tolerate failures of the peer components. Whenever its peer fails, the component continues the task until it has spent the allowed energy budget. In a similar fashion, modifications to the control loop information flow at run-time is supported; components can enter and leave the system at run-time. During the switching process, the component enters a temporal local autonomy state, attempting to finish its task within the energy budget.

Although, the methodology was implemented for a specific example, it is general and it can be applied to any kind of (nonlinear) physically based control. PL does not use any assumptions about the control strategy or the controlled system. Therefore, it can be applied to other systems with minor modifications. It also has been shown that the prerequisite for correct energy estimation can be relaxed at the cost of system performance. Even a rough energy estimate is sufficient to exploit the advantages of applying a PL.

Architecturally, the PL is similar to fault-tolerant control, as it consist of components that detect what would be non-passive (unstable) behaviors and correction procedure to ensure that behavior is passive. Yet, these components are separated from the control law, and this architecture allows to build control strategy oblivious to existence of the PL, which is only used as safety mechanism. Such architectural pattern does not restrict application of the PL.

The control strategy presented in here is oblivious to the energy state or the energy constraints. The performance of the system can be greatly enhanced if the control strategy incorporates information about energy states, for example via model predictive control techniques.

5

Conclusions and Recommendations

The goal of this work as formulated in Section 1.4 is to study ways to improve reliability and robustness of a robot through improving dependability of its motion control software. We have identified three threats to the motion control software: the quality of software implementation, the external faults from a connection to the physical domain such as failures of a sensor, and the external faults from a connection within the cyber domain such as communication to other components (Figure 5.1). A possible way to counteract each of these threats has been studied and presented in a corresponding chapter of this thesis.

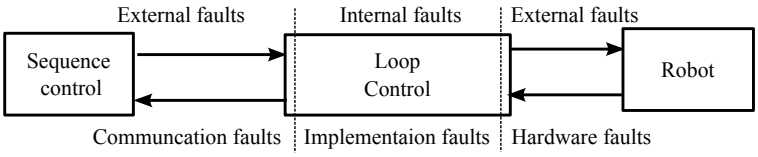


Figure 5.1: Threats to motion control software

The research conducted to find the answers to the questions posed in this thesis has opened new questions to be answered. This chapter is used to summarize the conclusion and contributions, and list some possible future work on the topics discussed in this thesis. This is done in three sections to address each of the three topics covered in this thesis.

5.1 Fault avoidance in the development process

A software development methodology advocated in Chapter 2 contributes to reliability of software components through fault avoidance means. This methodology is built on the concepts of software modeling and separation of concerns. The combination of these concepts ease the production of high quality software by reducing the number of attention points required from a developer.

In the chapter, we emphasized the need to model the software from different perspectives. These perspectives, in fact dictated by the separation of concerns, are supple-

mentary to each other and are required to create a complete software component. However, each perspective can be modeled separately allowing developers to focus their attention on one problem at the time.

Modeling the Computation to describe the desired algorithm allows to improve the quality of the algorithmic part of the developed software. In our vision, a developer can chose to sacrifice the freedom of implementation of an individual software component to attain high reliability of that component. We have indicated that modeling the Computation can be done using a set of well chosen modeling languages and that the flexibility of a complete application is retained by combining the components, described using different modeling languages.

The tool chain, implemented as a proof of concept, is used to demonstrate the possibility of modeling a software system using the advocated methodology. An appropriate tool support is an essential part of the methodology. Absence of such support hinders this methodology as the manual transformation of the models into the software component will lead to implementation faults.

Future work

Refinement of the developed tool-chain prototype is suggested future work. The tool-chain is developed as proof of concept and for use in future projects, it should be more extensively tested and its robustness should be improved.

Moreover, the tool chain currently supports only two frameworks and restricted set of languages for modeling the Computation. Adding support for more modeling languages would provide more freedom to the algorithm developer to chose a suitable language for each component. Adding support of more software frameworks will provide the freedom to system integrators.

The developed tool chain requires creation of separated components for each modeling language being used. In some cases, it is necessary to combine two models with in a single component. The template of Generic Architecture Component (GAC) presented by Bezemer (2013) demonstrates the need to combine Coordination and Computation primitives in a single component. The tool chain can be modified to accommodate that requirement.

A possible starting point to address these issues can be drawn from the partial code generation (for example discussed by Broenink et al. (2010)) which is already used in the tool chain. The partial code generation can be used to selectively enrich the software component with the details described by corresponding model or software framework. Alternative approach involves use of codels and their descriptions (drawing on the discussion by Mallet et al. (2010)), which are used to compose the software component in the final code-generation step.

5.2 Fault tolerance in mobile manipulators

The control strategies studied in Chapter 3 contribute to robust autonomy of a mobile manipulator by improving fault tolerance of control software with respect to external hardware faults. We discuss a model-based fault detection and present an approach to

mask typical robotic faults using existing redundancy. The discussed algorithms can be used to improve a robot's ability to fulfill designated task.

Using compliance control, we have demonstrated that fault tolerance can be improved even with existing hardware of the youBot, such as a wheel drive or sensor failure can be tolerated. Moreover, we showed that fail-safe actuation of the arm joints can be used to tolerate failures of corresponding motors or sensor. In both cases, the robot can maintain a small deviation on chosen control quality measure, *i.e.* the control precision.

Future work

Future work in the direction of fault-tolerant control consist of two steps: validation of the proposed method and re-use on different robots.

A validation of the algorithm performance on the youBot is recommended as future work. The validation can be performed in the same way as the verification was performed by applying fault injection technique. It is only possible to validate the FDI method and recovery procedure for the base wheel actuator/sensor failure, as existing hardware does not allow recover from failure of arm joint actuator/sensor failure. The validation process will be simplified due to the fact that controller part of the proposed method is already has been used in use case described in Chapter 1. Moreover, the model used during verification of the method can be used as Computation models and the methodology advocated in Chapter 1 can be applied.

The future work lies in re-use of this algorithm on another platforms such as LWR (Albu-Schäffer et al., 2007). Since this manipulator has fail-safe actuation, the proposed control strategy can be applied and the revocery procedure for the arm joint actuation failure can be validated. Another possibility is to extend the algorithm of fault-tolerant control to a pseudo omni-directional base. A pseudo omni-directional base use actuated caster wheels, e.g. the bases that are used in PR2 or Care-o-Bot. Current work presented here relies on Mecanum-wheels to allow omni-directional movement.

5.3 Passivity and Fault tolerance in distributed architectures

The Passivity Layer (PL) approach discussed in Chapter 4 has been devised to improve fault tolerance with respect to faults in communication between components. Inclusion of the PL as a part of safety mechanism introduces not only a way to cope with communication failure, but also a way to address safety of robot-environment interaction. Use of PL can significantly contribute to an ability of a robot for safe and reliable cooperation and interaction.

To study application of PL to improve reliability of distributed control, the communication faults have been treated from a robust-control point of view, where the faults are perceived as variations in the system parameters. A fault-tolerant control, similarly to robust control, seeks to maintain maximum performance of the control in presence of variations in the controlled system. The proposed PL ensures passivity of the control system. Depending on amount of faults/unexpected variations of parameters the PL smoothly reduce performance of the control by correcting the control signal

to preserve passivity. PL allows to maintain high performance of control around expected operation point and prevents unstable/dangerous behaviors due to parameter variations.

Future work

The structural and parametric design of the energy estimate block (Figures 4.1 and 4.3) represents the most important future work towards application of the passivity layer. We have shown that even a simple estimation of the needed energy could be sufficient for completing task. However, quality of energy estimate has significant effect on controller performance and identification of the non-passive behaviors and this will be a major research line for this framework.

Another direction of future work is using PL for safety purposes by monitoring of energy exchange with the environment, in other words, the amount of mechanical work performed by the robot. Inflicting injuries or mechanical damage to the equipment requires additional unexpected mechanical work (Laffranchi et al., 2009). PL provides an algorithmic solution to identify undesired and potentially dangerous energy exchange and enforce the energy constraints. Additional energy-based constraints for safety purposes can be derived from work on physical human robot interaction (Haddadin et al., 2008a,b; De Santis et al., 2008).

One more possible research direction of research for the use of the PL is including energy awareness provided by the PL into control strategies. PL is based on one step ahead prediction of energy exchange, which is sufficient for the purpose of maintaining passivity as a safety measure protecting the system from communication failure. However, predicting energy exchange over longer horizon and incorporating energy-based constrains into control signals computation using model-predictive control methods can be studied as it will provide important information about effects of PL on various performance measures for control.

5.4 Final word

High-quality motion control software complimented with fault-tolerance algorithms is essential for building autonomous robots designed for applications in an open environment. Achieving fault-tolerant high-quality motion control software was the goal of this work, and the presented combination of the development process and the fault-tolerance algorithms address this goal.

The failures of actuators, sensors, control and communication sum up to 88% of all robot failures, following the study of Carlson (2004) mentioned in Section 1.3. The combination of development methodology and algorithms address these type of failures, either by fault avoidance in development process, or by reducing their effect on robot performance through fault tolerance.

Further evaluation of the discussed combination of development methodology and algorithms by applying them to other robots is the overall recommendation for further work. Re-using the presented algorithms and development methodology will allow to refine them from a prototype stage to a product stage.

A

Modelling of the youBot dynamics

The modeling of the dynamics of a complicated robotic systems is a time consuming process. One of the ways to increase speed of development is reuse. This concept is rarely applied in modeling of robotic systems, because the mechanical parts of a robotic systems often require development of the dynamic models from scratch. Moreover, different applications require different types of modeling assumptions, even for the same robot. The concept of power-based modeling creates the possibility of representing a dynamic model in a simple way, while preserving the information about modeling decisions. This allows simple extension of the model. A bond graph is a graphical representation method of power-based modeling. Bond-graph models can be reused elegantly, because bond graphs are non-causal. The submodels can be seen as objects with defined power interfaces; bond-graph modeling is a form of object-oriented physical system modeling.

Bond-graph theory and notation are well developed and described in Breedveld (2004, 2008). In short, the representation is done by means of a directed graph: the vertices are submodels that describe interesting physical behavior and the edges represent energy relations. In addition to these edges, signals are used to pass information between vertices and do not represent energy exchange in the system. Bond graphs are a domain-independent representation of the model, therefore allowing easy capture the behavior of robotic systems, which often combines electrical, mechanical and hydraulic domains. In this paper, we will only concentrate on the mechanical behavior and only indicate how the model can be extended to other domains.

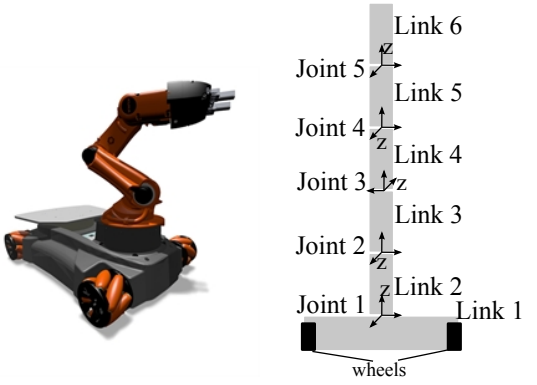
In order to develop a well structured model, the system behavior should be decomposed into concepts, idealized descriptions of physical phenomena. The concepts are combined into recognizable dominant behaviors of the tangible system parts (components). The components are used to compose the model of the robotic system.

The dynamic behavior of the youBot (mobile manipulator robot) can be decomposed, without losing generalization, into three types of components: Mecanum wheels, joints of the manipulator and links of the manipulator.

[1] This work has been reported in SIMPAR2010 Workshop proceedings, in Darmstadt, Germany (Dresscher et al., 2010).

Description of the platform

The modeling of the robotic platform youBot (Figure A.1a) is used as an example which can be extended to more complex mobile robot. The kinematic structure of the youBot robotic manipulator is shown in Fig. A.1b.



(a) Appearance **(b)** Schema of the youBot

Figure A.1: The youBot

The robotic manipulator has six links connected by 5 actuated rotational joints. The axis of rotation of joints is the z-axis in the frames depicted.

Four Mecanum wheels are mounted to the first link of the robotic actuator to make it mobile, this is shown in Fig. A.2.

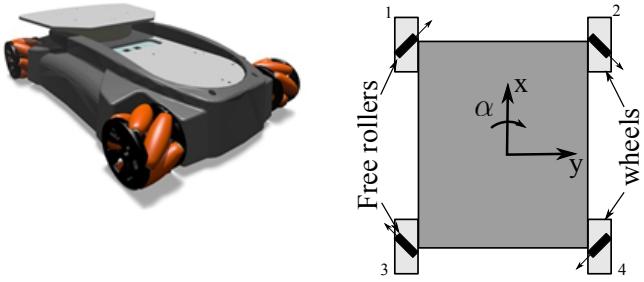
Submodel of a link

A modeling process begins with defining a dominant behavior of the component. For robotic applications, the dominant behavior of the links in the manipulator is the behavior of a body to which external wrenches are applied. As long as the youBot is used within the specifications, it is valid to assume that the links are rigid and are treated as rigid bodies. However, the submodel can be adapted to include other behavior without effecting the interface with other parts of the youBot model.

Modeling rigid body dynamics of a link

The model of rigid-body dynamics consists of inertia and gyroscopic effects. According to screw theory the kinematics of a rigid-body motion can be represented as a rotation (ω) about an axis along with translation (v) along the same axis (Ball, 1900). Furthermore, we will use the following notation:

- $T = \begin{pmatrix} \omega \\ v \end{pmatrix}$: twist



(a) youBot Base

(b) Schematic representation of the youBot base

Figure A.2: The youBot base with Mecanum wheels

- $W = \begin{pmatrix} \tau \\ F \end{pmatrix}$: wrench
- $I = \begin{pmatrix} J & 0 \\ 0 & M \end{pmatrix}$: inertia tensor.
- F: force
- P: momentum screw
- τ: torque
- M: mass matrix
- J: inertia matrix

The wrench balance representing the inertia effects, expressed in the principal inertia frame k, (in the center of gravity and oriented along the three primary inertia axes) will have a form (Stramigioli and Bruyninckx, 2001):

$$I^k \dot{T}_a^{k,0} = \begin{pmatrix} \tilde{P}_\omega{}^k & \tilde{P}_v{}^k \\ \tilde{P}_v{}^k & 0 \end{pmatrix} T_a^{k,0} + (W^k)^T \tag{A.1}$$

where

- k is the principal inertia frame of the body
- 0 is the inertial frame
- a is the body

In the above relation the component for an inertia can be recognized on the left hand side of the equal sign. The first component at the right hand side of the equal sign represents the fictitious forces and torques (wrenches) including the gyroscopic effect, the second component at the right hand side of the equal sign represents externally applied wrenches. A wrench balance as described by the formula above is represented in a bond graph by a 1-junction. The bond graph shown in Fig. A.3 represents the equation above.

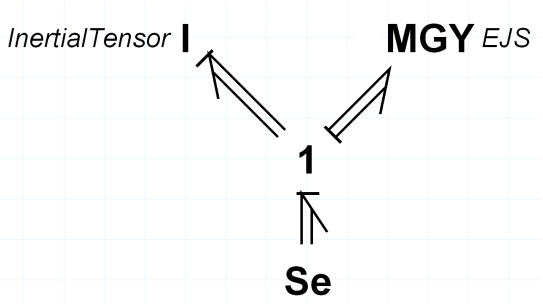


Figure A.3: A bond-graph model of a rigid body including gyroscopic effects

In this model the fictitious wrenches including the gyroscopic effects are expressed by a gyrator (MGY element). A gyrator represents a power continuous relation between efforts and flows. The externally applied wrenches are represented by the Se (Source of effort) element.

Externally applied wrenches

The externally applied wrenches include joint reactions, collisions, gravity etc. The most significant ones, that have a constant effect on the link dynamics, are the joints reactions and gravity. The joints will be connected to a link by means of power ports, through which the joints apply wrenches to the links. However, these wrenches are expressed in the frame that corresponds with the actuated joint. The wrench balance, as discussed in the previous section, is represented in a principal inertia frame of the body. Therefore, a transformation of coordinates is required between these parts of the model.

A homogeneous matrix H_i^j represents the position of a frame i with respect to a frame j with a translation component p and a rotation component R:

$$H_i^j = \begin{pmatrix} R_i^j & p_i^j \\ 0 & 1 \end{pmatrix} \tag{A.2}$$

A twist, in matrix form (Stramigioli and Bruyninckx, 2001), is defined as the product of a position and the derivative with respect to time of the same position (Stramigioli

and Bruyninckx, 2001):

$$\begin{aligned} \tilde{T}_i^{i,j} &= H_j^i \dot{H}_i^j \\ \tilde{T}_i^{j,j} &= \dot{H}_i^j H_j^i \end{aligned} \tag{A.3}$$

These two expressions can be combined into:

$$\tilde{T}_k^{j,l} = H_i^j \tilde{T}_k^{i,l} H_j^i \tag{A.4}$$

Which represents a change of coordinates for twists in matrix form. It is possible to see (Selig, 1996; Stramigioli and Bruyninckx, 2001) that a change of coordinates for twists in vector form, as applied in this paper, can be expressed as:

$$T_k^{j,l} = Ad_{H_i^j} T_k^{i,l} \tag{A.5}$$

with

$$Ad_{H_i^j} = \begin{pmatrix} R_i^j & 0 \\ \tilde{p}_i^j R_i^j & R_i^j \end{pmatrix} \tag{A.6}$$

the adjoint of the homogeneous matrix H_i^j.

Since a change of coordinates is power continuous, an expression for a change of coordinates for wrenches can be found:

$$W^j T_k^{j,l} = W^j Ad_{H_i^j} T_k^{i,l} = (Ad_{H_i^j}^T W^{j\,T})^T T_k^{i,l} = W^i T_k^{i,l} \tag{A.7}$$

such that

$$(W^i)^T = Ad_{H_i^j}^T (W^j)^T \tag{A.8}$$

Since the relations are power continuous, a TF element is used to represent this change of coordinates in the bond-graph language.

Coordinate transformations are required between three frames:

1. The body-fixed frame in the previous joint, from now on called frame i
2. The body-fixed frame in the next joint, from now on called frame j
3. The principal inertia frame of the link, from now on called frame k.

To relate these three frames only two changes of coordinates are required since the third change of coordinates can be composed of the other two. For example:

$$H_j^k = H_i^k H_j^i \tag{A.9}$$

In equation(A.9) the two changes of coordinates changes are applied between:

1. Frame *i* and frame k
2. Frame *j* and frame i

And frame *i* has a central role. The choice can be made to give either frame i, frame *k* or frame *j* this central role. This does not effect the model behavior or reusability of the model. In this paper, frame *i* has the central role. Therefore, the changes of coordinates are applied as shown in the example.

Since deformation of the link is neglected, it is assumed that the above mentioned changes of coordinates are constant in time. However, the model can easily be extended to include those effects by replacing the linear relation with non-linear, time-dependent ones. Fig. A.4 shows the model of the link including the discussed changes of coordinates. *p* and *p1* are the power ports to which the joints will be connected.

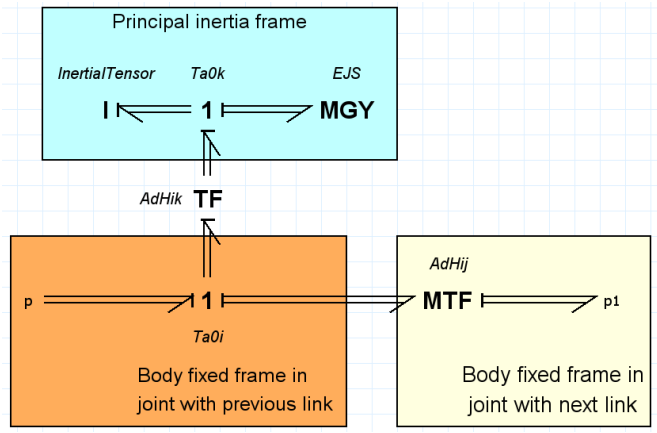


Figure A.4: A bond-graph model of a rigid body with changes of coordinates to the joints

Gravity, expressed in the inertial frame 0, is a constant force (wrench) that applies to the body in the negative z-direction and can be modelled as a Se element in the bond-graph language. The other parts of the model are not expressed in the inertial frame and therefore a transformation of coordinates is required between the inertial frame and one of the other frames. Since the frame in the previous joint is used as a starting point, it is chosen to apply the coordinate transformation between system 0 and frame i. The H-matrix that corresponds to this change of coordinates is passed on by the previous joint as a signal that modulates the transformations.

Fig. A.5 shows the complete submodel of a link. In this model, the Se element representing gravity can be found with the coordinate transformation between the inertial frame 0 and frame i. The signal port Hin is used to obtain the position of frame *i* with respect to the inertial frame. This is then multiplied by H_i^j, the relative position of frame *j* with respect to frame i, to obtain the position of frame *j* with respect to the inertial frame. This position is passed on the the next joint through signal port Hout. The ports *p* and *p1* are the power ports of the link, to which the joints will be connected.

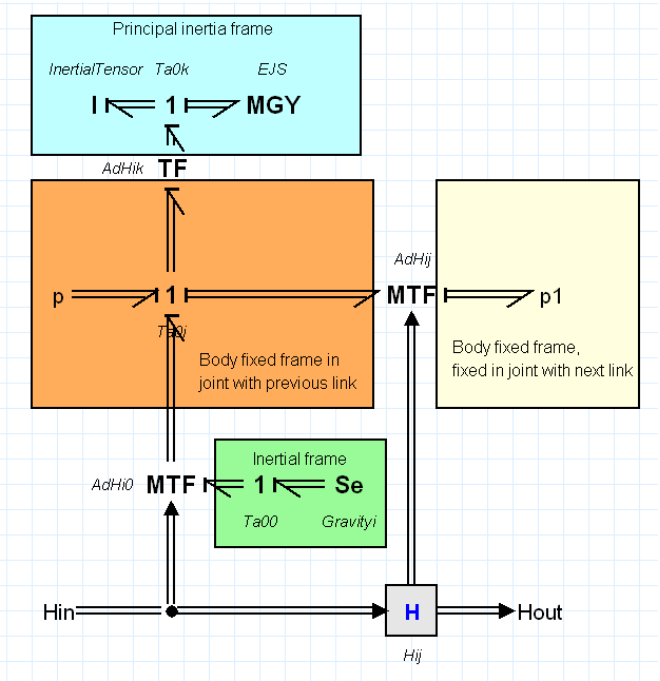


Figure A.5: A bond-graph model of a link

Submodel of a joint

Just as with the submodel of a link, it has to be decided which behavior will be modelled and which behavior will be neglected. For this model, it is decided only to model the transfer of energy in a joint. Therefore, the following behavior has explicitly been neglected:

- Friction in the joint
- Energy storage in the joint

However, the model can easily be extended to include these and other effects.

Since a joint establishes an energetic connection between two links, it imposes a relation between the wrenches and the twists of the two bodies.

A wrench applied in a joint is applied between the two connected links and, in fact, applies the same wrench to both links. This specifies the relation between the wrench applied by the actuator and the wrenches applied to the connected links:

$$W_{act} = W_{link1} = W_{link2} \tag{A.10}$$

On the other hand, there is a relation between the twists of link 1 with respect to the inertial frame, expressed in frame n ($T_1^{n,0}$) and the twist of link 2 with respect to the inertial frame, expressed in frame n ($T_2^{n,0}$). Just as with other flow type variables the following relation holds:

$$T_2^{n,0} = T_1^{n,0} + T_2^{n,1} \tag{A.11}$$

The above two relations are represented by a 0-junction in the bond-graph language. For these relations to apply, all twists and wrenches should be expressed in the same frame. For this paper frame i is selected, this does not effect reusability of the model. A joint is an energetic connections between two links. The twists and wrenches of these two links are expressed in frames fixed with the link. As a result they are expressed in different frames and a change of coordinates is required.

In the relations of the joint the actuation is a wrench, while the actuator applies a torque in the joint. To transform between the actuator torque and the wrench applied between the two bodies and on the other hand, a unit wrench can be applied:

$$W = \hat{W}\tau \tag{A.12}$$

Where, in case of a rotational joint:

$$\hat{W} = \begin{pmatrix} \hat{\tau} \\ 0 \end{pmatrix} \tag{A.13}$$

and $\hat{\tau}$ is the axis around which the torque is applied.

Due to power continuity this imposes a relation between the twist in the joint and the joint velocity:

$$W^T T_a^{i,b} = (\hat{W}\tau)^T T_a^{i,b} = \tau \hat{W}^T T_a^{i,b} = \tau \dot{q} \tag{A.14}$$

such that

$$\dot{q} = \hat{W}^T T_a^{i,b} \tag{A.15}$$

Since this set of relations is power continuous, they can be represented by a transformer in the bond-graph language. Fig. A.6 shows the model of a joint.

The change of coordinates is represented by the MTF element "*Adji*". The conversion from joint torque to wrench and from twist to joint velocity is represented by the TF element "*uTbai*".

In the bond-graph model two other blocks are present, an $\int$-block and an x-block. The $\int$-block calculates the joint position based on an integration of the joint velocity and uses this to construct the H matrix that represents the position of the joint: H_j^i. This matrix is then used for the coordinate transformation. A multiplication block indicated by the symbol x multiplies two matrices. The two H-matrices that are multiplied are:

- The position of frame i with respect to the inertial frame 0, obtained from Hin (H_i^0)

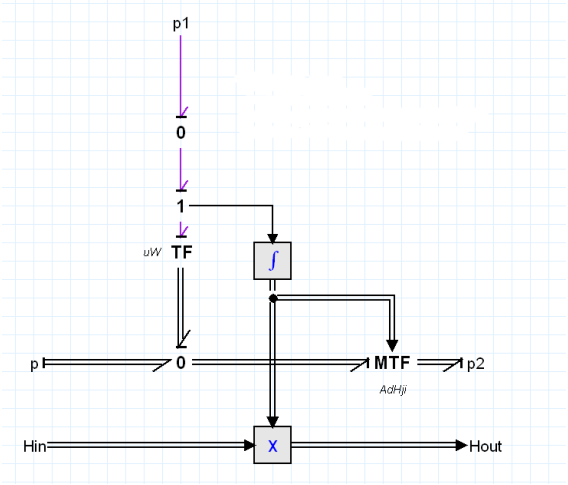


Figure A.6: A bond-graph model of a joint

- The position of frame j with respect to frame i (H^i_j)

such that the position of frame j with respect to the inertial frame is obtained:

$$H^0_j = H^0_i H^i_j \tag{A.16}$$

Submodel of a Mecanum wheel

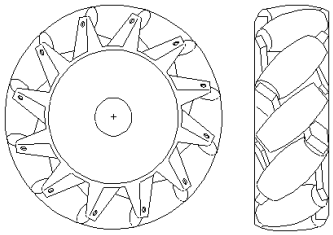


Figure A.7: An example of a Mechanum wheel (Source: www.robotika.sk)

A Mechanum wheel is a wheel with rollers mounted on it. These rollers are mounted in an angle of 45° with respect to the wheel and can freely rotate. Fig. A.7 shows an example of a Mecanum wheel.

In this paper a Mecanum wheel is considered as a construction that enables the application of forces to the robot by actuation of the wheel axis. All dynamic behavior of the Mecanum wheel is neglected for simplicity reasons.

A Mechanum wheel is, first of all, a wheel. Therefore, the model of a wheel is first discussed.

Model of a wheel

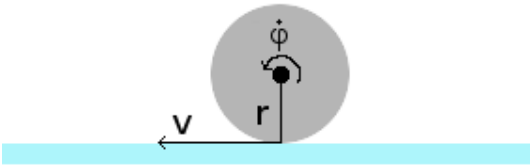


Figure A.8: A common wheel represented as transformer

An ideal wheel on a surface, as shown in Fig. A.8, creates a relation between the rotational velocity ($\dot{\phi}$) of the wheel and the translational velocity (v) of the wheel with respect to the surface. It is common knowledge that the radius (r) of the wheel is a measure for this relation. More specifically:

$$v = r\dot{\phi} \tag{A.17}$$

On the other hand, a wheel also imposes a relation between the torque (τ) applied to the wheel and the force (F) between the wheel and the surface. The radius also is a measure for this relation:

$$F = (1/r)\tau \tag{A.18}$$

When comparing the energy flow, or power, on the two sides (power ports) of the wheel (translation and rotation) it is clear that it is equal:

$$P_{translation} = vF = r\dot{\phi}(1/r)\tau = \dot{\phi}\tau = P_{rotation} \tag{A.19}$$

In the bond-graph language this ideal behavior is described by the standard element transformer.

Extension of the model of a wheel to the model of a Mecanum wheel

At the contact with the surface, a Mecanum wheel has rollers where a common wheel has none. These rollers can rotate freely around one axis and are mounted in an angle of 45° with respect to the transformation direction of the wheel. Fig. A.9 clarifies the position of a roller (grey, filled rectangle) in the wheel (black rectangle outline). In Fig. A.9 two frames are shown:

1. One frame that is aligned with the wheel, this frame will be called frame 1
2. A second frame that is aligned with the roller, this frame will be called frame 2

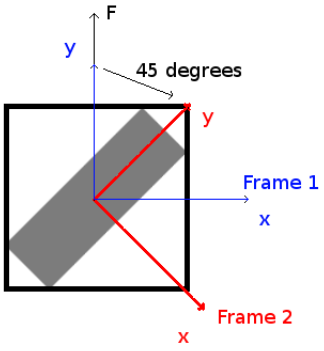


Figure A.9: Forces in frame 1 aligned with the wheel (blue) and frame 2 aligned with the roller (red)

In Fig. A.9 the axis around which the wheel rotates is the x-axis of frame 1 and the axis around which the roller can rotate is the y-axis of frame 2.

Since the roller can rotate freely around its y-axis, it acts as a bearing for translational movements in its x-direction. Just like in a common bearing the friction is minimized, for this model it is assumed that no friction is present. As a result of the lack of friction in this direction, there is no relation between:

1. the force applied by the wheel/ velocity of the wheel and
2. the force applied between the wheel and the surface/ velocity of the wheel with respect to the surface

in the x-direction of the roller. Any force applied in this direction will only contribute to the rotation of the wheel. When putting this in perspective with the model of the wheel discussed earlier, there is a transformation ratio of 0 in this x direction of frame 2. Note that this is expressed in a different frame than the transformation of the wheel, which is expressed in frame 1.

To model the effect of the rollers in the bond-graph language, a 2-dimensional transformer element is used to represent the effect of the roller in frame 2. The transformation ratio (r) of this transformer is:

$$r = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{A.20}$$

It has a transformation ratio of 0 in the x direction and a transformation ratio of 1 in the y directions. Note that this transformer represents a relation between two translational components.

Since the transformation of the wheel is expressed in frame 1 and the transformation of the roller in frame 2, one of the two should be expressed in the frame of the other.

Such that, both transformations are expressed in the same frame. It is chosen to express the transformation of the roller in frame 1.

To express a transformer in a different frame, the transformation ratio should be pre-multiplied by the change of coordinate matrix A and post-multiplied by its inverse:

$$TF^j = A_i^j TF^i (A_i^j)^{-1}, \text{where} (A_i^j)^{-1} = A_j^i. \tag{A.21}$$

In the bond-graph language, this change of coordinates can be represented by transformers as shown in Fig. A.10.

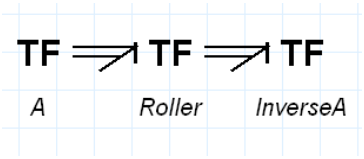


Figure A.10: Change of coordinates for a transformer in the bond-graph language

In case of a pure rotation (Stramigioli and Bruyninckx, 2001):

$$A = \begin{pmatrix} cos(\phi) & -sin(\phi) \\ sin(\phi) & cos(\phi) \end{pmatrix} \tag{A.22}$$

Where ϕ is the rotation between frame *i* and frame j.

When going from frame 2 to frame 1 there is a rotation of -45°, so in this case ϕ equals -45°.

Now the bond-graph model for the complete Mecanum wheel can be constructed as shown in Fig. A.11.

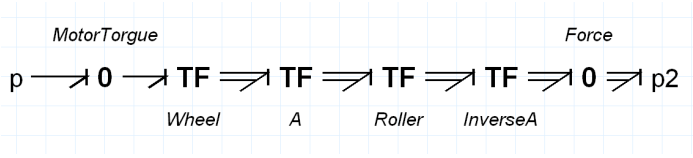


Figure A.11: The complete bond-graph model of a Mecanum wheel

Please note that in practise two types of Mecanum wheels are used:

1. One where the rollers are mounted in an angle of 45° with respect to the wheel and
2. a second type where the rollers are mounted in an angle of -45° with respect to the wheel

This difference results in slightly different models. However, both models are structured as explained in this section.

Construction of the model

The model of the robotic manipulator is constructed as a serial link in an object oriented way. In this model, two of the previously discussed submodel types are used:

1. submodels of links, connecting two joints
2. submodels of joints, connecting two links

Such that the model of the robotic arm is as shown in Fig. A.12.

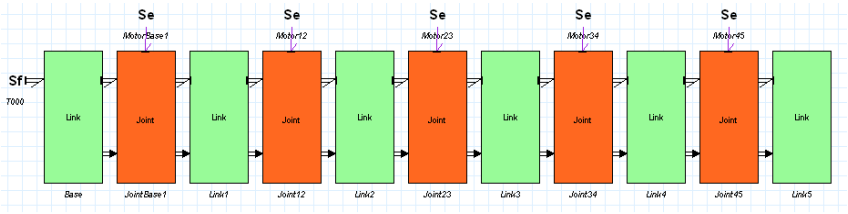


Figure A.12: Model of a robotic arm

The Se elements connected to all the joints represent the torques applied by the actuators in the joints. If necessary, the model can be extended with submodels of the electric motors. Furthermore, two connections can be seen between all joints and links:

1. a power connection establishing energetic coupling of the elements
2. a signal connection that passes on the homogeneous matrix representing the position of the connection point

So, every link and joint passes the position of the connection point with the next element on to the next element.

Connection of the Mecanum wheels to the robotic manipulator

As described in section A, four Mecanum wheels are connected to the first link of the robotic manipulator. However, in the model they cannot be connected directly, since the model of the robotic manipulator uses screw theory, while the submodel of the Mecanum wheel does not. The model of the Mecanum wheels provides a linear force caused by the Mecanum wheel as a result of the axis actuation. This linear force is applied in a point of the link. If this would be transformed to a wrench, the wrench would be applied to the body. This is a clear advantage in addition to the earlier mentioned advantage of being able to address rotations and translations simultaneously.

Transformation from a force in a point to wrench applied to a body and from a twist of a body to a velocity of a point is power continuous: the operation does not add energy to or remove energy from the system. Transformation of a twist to a linear velocity in a point (p) is given as:

$$\dot{p} = \omega \wedge p + v \tag{A.23}$$

Where a twist has the following shape:

$$T = \begin{pmatrix} \omega \\ v \end{pmatrix} \tag{A.24}$$

This relation can be expressed in matrix form:

$$\dot{p} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} = \begin{pmatrix} 0 & z & -y & 1 & 0 & 0 \\ -z & 0 & x & 0 & 1 & 0 \\ y & -x & 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} \omega \\ v \end{pmatrix} \tag{A.25}$$

Based on power continuity, the relation between a force and the corresponding twist can be found:

$F^T v = F^T AT = W^T T$ so $F^T A = W^T$

when transposed on both sides :

$A^T F = W$

These two relations together are the relations of a transformer in the bond-graph language, as shown in section A. So, a transformation from screws to linear velocities/ forces and vise-versa can be accomplished with a multi-dimensional transformer in the bond-graph language. Please note that the transformation transforms between a three dimensional force/ velocity and a wrench/ twist. The Mechanum wheels exert no force in the z-direction and therefore the force vector is extended to:

$$F = \begin{pmatrix} F_x \\ F_y \\ 0 \end{pmatrix} \tag{A.26}$$

With all the forces applied by the wheels transformed to wrenches applied to the link, they can be summed. In the bond-graph language, this is represented by a 1-junction. The summed wrench can then be applied to the base. Fig. A.13 shows the complete model of the Mecanum wheels connected to the first link of the robotic manipulator. The blocks "Mecanum wheel type A" and "Mecanum wheel type B" are submodels containing the model of a Mecanum wheel as discussed earlier. The Se elements are sources of effort, they apply a fixed torque to the axis of the Mecanum wheels.

Since the position of the base is required in the base block, an $\int$-block has been added. This block calculates the position of the block based on the current position and the twist of the base. Furthermore, an R element (resistance) and a C element (stiffness) can be found in the model. These elements only act in the z-direction of the base and represent the contact with the floor. Since the floor is very stiff and has a high resistance, the values of these two components are high. Without these elements, the robot-model will accelerate in the negative z-direction due to gravity.

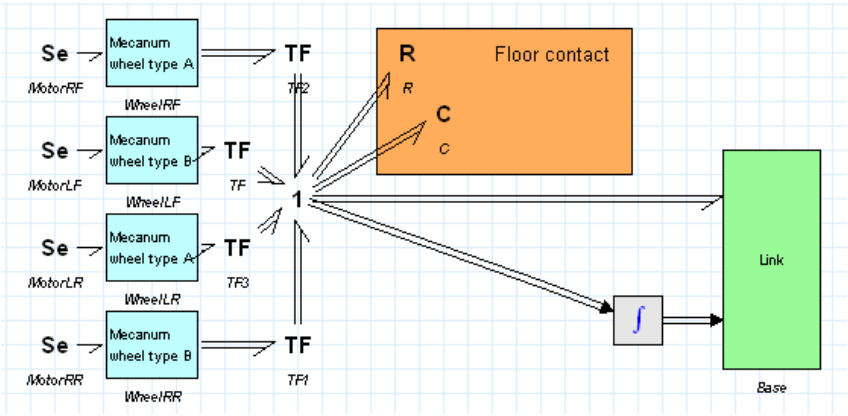


Figure A.13: Model of the Mecanum wheels connected to the first link

B

Components of youBot motion stack

This appendix contains details of the components used in motion stack in Chapter 2. The overall structure of the motion stack presented in Figure 2.14.

Actuator control

The firmware of the youBot provides three options to access the actuators: position control, velocity control and torque (current) control. Compared to the generic motion-stack architecture (Figure 2.13), the firmware spans two levels of control: actuator and joint control. Due to several limitations imposed by the firmware (version 1.48), only actuator control from the firmware can be used. The foremost reason is that switching between joint-control types (e.g. from torque control to position control) breaks the real-time control constraints. The firmware performs control on each joint independently thus no gravity compensation is present and configuration dependent inertia changes are not taken into account.

Hardware drivers are implemented to supply a uniform manner of access to the hardware-device interfaces and to raise the level of abstraction to component layer as recommended by Kraetzschmar et al. (2010). The youBot drivers reflect the firmware functionality (Locomotec, 2013). The drivers are integrated into the *youBot_OODL* component and provide another level of abstraction for the motion stack. The *youBot_OODL* component accepts as an input an array of values that represent the desired set-points. The configuration of the component determines, per joint, the type of the control (position control, velocity control and torque (current) control). The set points are passed to the firmware and transformed into analog control signal using zero-order hold.

Joint control

The joint control presented here uses the torque control of the *youBot_OODL* component. Three components are responsible for joint control:

- *ArmPoseController* - PD controller pose and gravity compensation as joint control;

[1] This work has been reported in public deliverable D6.2 of BRICS project (Brodskiy et al., 2013).

- *BasePoseController* - PD controller for the base (2D pose);
- *YouBotKinematics* - Forward kinematics and map of tool tip forces to joint torques.

Note, the position control of the individual joints has been re-implemented to avoid the need for switching the control type in the firmware, which would break real-time constraints.

The structure of *ArmPoseController* is shown in Figure B.1. It uses the *active_joint* map to switch on or off the PD action of individual joints. Two types of safety checks are implemented for the arm. Damage to the arm from crashing into the physical limits of the joints is prevented by soft joint limits. The soft joint limits restrict movement of the arm by acting as virtual springs around hardware joint limits. The second safety mechanism is the passivity layer discussed in Chapter 4.

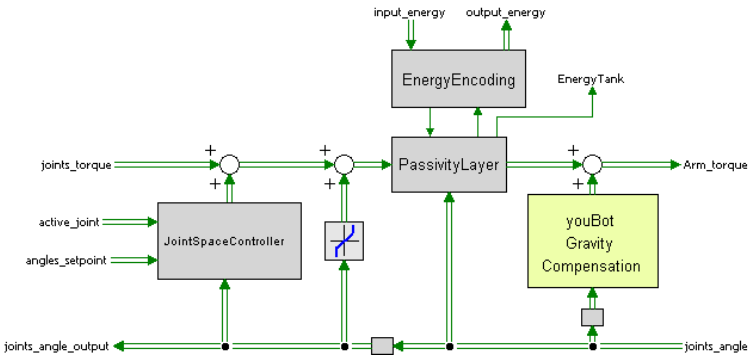


Figure B.1: ArmPoseController component

The structure of *BasePoseController* is shown in Figure B.2. Similar to the *ArmPoseController* the *active_joint* map to determine which degree of freedom is controlled by the PD action. The *BasePoseController* also includes an inverse kinematics for the youBot wheel base (*BaseTF* Figure B.2), as discussed in Chapter 3 implementing Eq. 3.7 and 3.4. This allows to map three degrees of freedom (in 2D) to the four wheels of the youBot.

Since both pose controllers include the passivity layer, other components can be connected over unreliable communication channels. This is exemplified in the overall architecture (see Figure 2.14) by a set of time delay blocks between the *YouBotKinematics* and the *ArmPoseController* and *BasePoseController* components. However, it remains essential for performance reasons that the pose controllers are serialized (i.e. executed in a specific order) with the driver component, as indicated in Section 4.2 it is DT-CT communication type. In the implementation of the use cases this is achieved by adding a scheduling component.

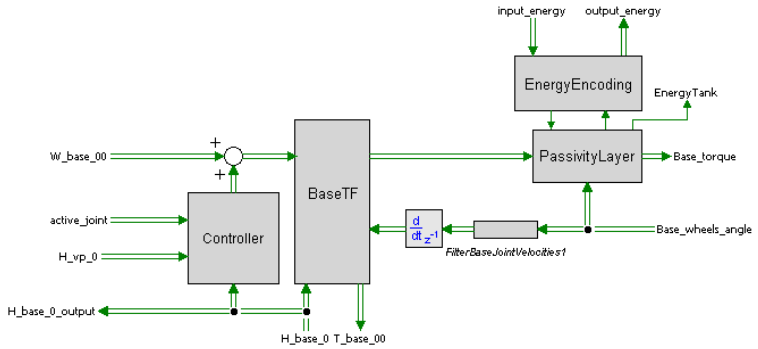


Figure B.2: BasePoseController component

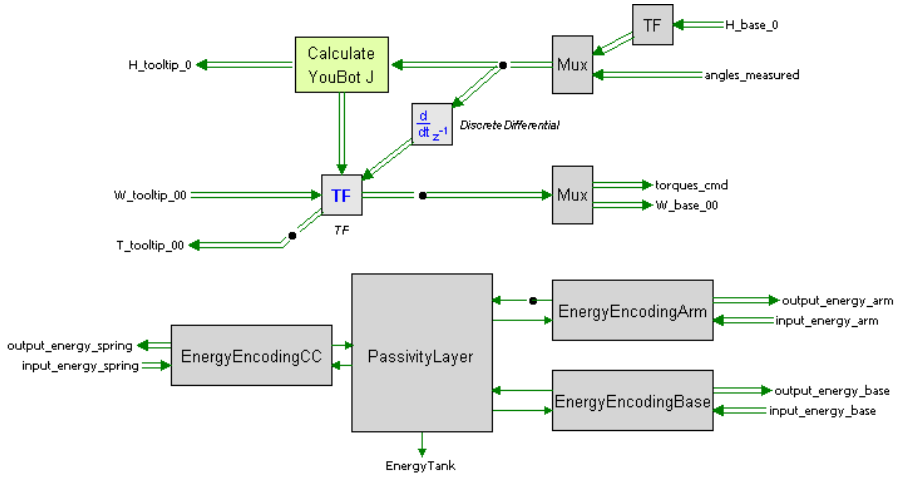


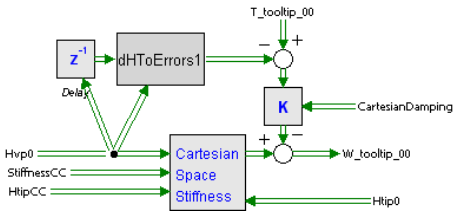
Figure B.3: YouBotKinematics component

The *YouBotKinematics* component is responsible for calculating the forward kinematics, and mapping tool tip wrenches to joint torques and joint angular velocities to tool tip twists as discussed in Chapter 3. Forward kinematics is computed with Brockett's product of exponentials formula Brockett (1984). The mapping between tool tip wrenches and joint torques is a geometric Jacobian. Thus, the components presented here can be used for controlling back drivable robots Hogan (1985).

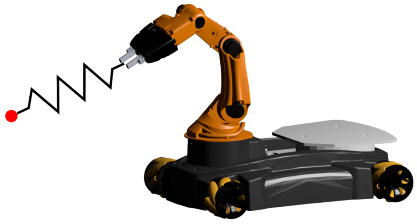
Additionally, the *YouBotKinematics* also performs energy routing to the pose controllers. The energy routing is used to equalize the energy levels in all connected components. Note that the placement of the energy routing is purely an implementation choice, since it could have been a separate component as well.

Cartesian control

The *CartesianController* component provides an impedance controller, as described in Chapter 3. The controller uses an elastic wrench for impedance computation Stramigioli (1998). The impedance is defined as a virtual spring, which is attached between the desired set-point and the tool-tip of the youBot. A schematic representation of the Cartesian controller is shown in Figure B.4b.



(a) Cartesian Controller component



(b) Schematic representation of Cartesian Controller

Figure B.4: Cartesian control

The control of the tool-tip is obtained by varying the properties of the spring, such as the rest length or the center of stiffness. In case of the rest length property, it is changed from the current position/orientation of the tool-tip to the desired ones. The resulting force/torque will be in the direction/orientation of the least potential energy, i.e. towards the desired tool-tip position. The center of stiffness is the transformation from the tool-tip of the robot to the point where the stiffness parameters are defined. The center of stiffness allows to define any type of an impedance active on the tool-tip with 9 parameters Loncaric (1987). This stiffness consists of 6 anisotropic stiffnesses (one for each degree of freedom) and 3 coupled stiffnesses. The anisotropic stiffnesses define direction-dependent behaviours for the tool-tip. This can for example be used to solve a peg-in-the-hole problem by emulating a remote center of compliance device. The coupled stiffnesses define correlations between rotational and translational displacements, which can be applied for operating a crankshaft or to screw in screws Lon-

caric (1987). Both properties can be varied for the impedance controller defined in the *CartesianControl* component, see Figure B.4a and 2.14.

Trajectory interpolator and planner

The trajectory interpolator and planner in Figure 2.13 are shown in Figure 2.14 as set-point generators, since they are mainly a task-dependent part of the motion stack. For example, in the tele-operation use case the interpolator is replaced by the haptic device and the planning is done by the human. Whilst for the BRICkS stacking the interpolation is done via low-pass filtering of the set-point signal and trajectory planning is done by the Finite State Machine (FSM).

Within the passivity-based approach, an estimate of the energy that will be dissipated during a task's execution is a part of the planned trajectory. There are several different ways to determine such an estimate. For the use cases two different approaches are applied, based on the type of operation (autonomous versus tele-manipulation). In the tele-operation context, the energy estimation for moving the robot can be displayed to the operator using a haptic device Ryu et al. (2004). The haptic device presents the estimation result as a friction to the user, by exerting a force opposite to the users desired movement. For autonomous operation, the estimation is based on the model of the controlled robot and the expected type of interaction. The autonomous operation use cases utilize a (simple) estimate of the energy usage (*SimpleEnergyEstimate* component), based on experimentally obtained energy consumption during movement. Among others taking internal friction of the arm into account. Then the estimated consumption is used during task execution to smartly replenish the energy tank. It ensures the safety of the system by preventing combinations of high forces and high velocities, which can arise during any collision (i.e. at impact).

Although the trajectory interpolator and planner are mostly task-specific components in the motion stack, the energy estimation is a required part for the trajectory planner and the trajectory interpolator should monitor the power flow, i.e. the energy budget for a task in time. When the energy consumption is estimated correctly, it enables the passivity layer to ensure safety and passivity of interaction, without interfering with system performance.

C

Derivation of state-space representation of the case study system

This appendix provides derivation of Eq. 4.19, which used to make Figure 4.8

To perform stability analysis the methodology described by Bemporad et al. (2010) is used. It is required to obtain the state space representation of the system. Dynamic part of the system (Eq. 4.14, 4.17) can discretized as follows:

$$A_{dyn} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ \frac{-K_{S_1}}{M_{D_1}} & \frac{-B_{D_1}}{M_{D_1}} & 0 & 0 & \frac{K_{S_1}}{M_{D_1}} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-K_{S_2}}{M_{D_2}} & \frac{-B_{D_2}}{M_{D_2}} & \frac{K_{S_2}}{M_{D_2}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ \frac{K_{S_1}}{M_b} & 0 & \frac{K_{S_2}}{M_b} & 0 & -\frac{K_{S_1}}{M_b} - \frac{K_{S_2}}{M_b} & \frac{-B_b}{M_b} \end{bmatrix};$$

$$B_{dyn} = \begin{bmatrix} 0 & 1/M_{D_1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/M_{D_2} & 0 & 0 \end{bmatrix}^T.$$

$$F_{dyn} = e^{hA_{dyn}}; \quad G_{dyn} = \int_0^h e^{\tau A_{dyn}} d\tau B_{dyn}, \text{where}$$

$$u_k = \begin{bmatrix} F_{C_1} & F_{C_2} \end{bmatrix}^T;$$

$$x_k = \begin{bmatrix} q_{D_1,k} & \dot{q}_{D_1,k} & q_{D_2,k} & \dot{q}_{D_2,k} & q_{b,k} & \dot{q}_{b,k} \end{bmatrix}^T;$$

$$x_{k+1} = F_{dyn} x_k + G_{dyn} u_k \tag{C.1}$$

Combining equations C.1 and discrete version of equation 4.15 closed loop dynamics

of the system is obtained.

$$x_{k+1} = F_{sys}x_k + G_{sys}u_{sys,k}, \quad \text{where} \tag{C.2}$$

$$u_{sys,k} = \begin{bmatrix} q_{R_1} & q_{R_2} \end{bmatrix}$$

$$G_{sys} = G_{dyn}\begin{bmatrix} K_{P_1} & 0 \\ 0 & K_{P_2} \end{bmatrix}$$

$$F_{sys} = F_{dyn} - G_{dyn}K_{dyn}C_{dyn} \tag{C.3}$$

$$C_{dyn} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix};$$

$$K_{dyn} = \begin{bmatrix} K_{P_1} & K_{D_1} & 0 & 0 \\ 0 & 0 & K_{P_2} & K_{D_2} \end{bmatrix}$$

Adding extended state to maintain the delayed control inputs for n samples delay:

$$\xi_{s,k} = \begin{bmatrix} x_k & q_{R_1,k-1} & q_{R_2,k-1} & \cdots & q_{R_1,k-n} & q_{R_2,k-n} \end{bmatrix}^T \tag{C.4}$$

$$\xi_{s,k+1} = \underbrace{\begin{bmatrix} F_{sys} & 0^{6\times 2n} & G_{sys} \\ 0^{2\times 6} & 0^{2\times 2n} & 0^{2\times 2} \\ 0^{2n\times 6} & I^{2n\times 2n} & 0^{2n\times 2} \end{bmatrix}}_{F_{sys}} \xi_{s,k} + \underbrace{\begin{bmatrix} 0^{2\times 2} \\ I^{2\times 2} \\ 0^{2n\times 2} \end{bmatrix}}_{G_{sys}} u_{sys,k} \tag{C.5}$$

To obtain forces commanded to the actuators following output matrix is used.

$$C_{sys} = \begin{bmatrix} -K_{dyn}C_{dyn} & 0^{2,2n} & \begin{bmatrix} KP1 & 0 \\ 0 & KP2 \end{bmatrix} \end{bmatrix} \tag{C.6}$$

The discretization of the virtual body simulation is obtained as:

$$A_c = \begin{bmatrix} -R_v/M_v & -K_v \\ 1.0/M_v & 0 \end{bmatrix}; \quad B_{c_1} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}; \quad B_{c_2} = \begin{bmatrix} Kv & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}; \tag{C.7}$$

$$C_c = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}; \quad D_c = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix};$$

$$x_c = \begin{bmatrix} \dot{q}_v & q_v \end{bmatrix}^T; u_{c_1} = \begin{bmatrix} F_{C_1} & F_{C_1} \end{bmatrix}^T; u_{c_2} = \begin{bmatrix} q_{vp} & L_{R_1} & L_{R_2} \end{bmatrix}^T;$$

$$F_c = e^{hA_c}; \quad G_{c_1} = \int_0^h e^{\tau A_c} d\tau B_{c_1}; \quad G_{c_2} = \int_0^h e^{\tau A_c} d\tau B_{c_2}$$

$$x_{c,k+1} = F_c x_{c,k} + G_{c_1} u_{c_1,k} + G_{c_2} u_{c_2,k} \tag{C.8}$$

$$y_{c,k} = C_c x_{c,k} + D_c u_{c_2,k} \tag{C.9}$$

Adding extended state to maintain the delayed control inputs for n samples delay:

$$\xi_{c,k} = \begin{bmatrix} x_{c,k} & F_{C_1,k-1} & F_{C_2,k-1} & \cdots & F_{C_1,k-n} & F_{C_2,k-n} \end{bmatrix}^T \tag{C.10}$$

$$\xi_{c,k+1} = \underbrace{\begin{bmatrix} F_c & 0^{2\times 2n} & G_{c_1} \\ 0^{2\times 2} & 0^{2\times 2n} & 0^{2\times 2} \\ 0^{2n\times 2} & I^{2n\times 2n} & 0^{2n\times 2} \end{bmatrix}}_{F_{cd}} \xi_{s,k} + \underbrace{\begin{bmatrix} 0^{2\times 2} \\ I^{2\times 2} \\ 0^{2n\times 2} \end{bmatrix}}_{G_{cd}} u_{c_1,k} + \underbrace{\begin{bmatrix} G_{c_2} \\ 0^{1\times 3} \\ 0^{2n\times 3} \end{bmatrix}}_{G_{cf}} u_{c_2,k} \tag{C.11}$$

$$C_{cd} = \begin{bmatrix} C_{cc} & 0^{2\times 2n} & 0^{2\times 2} \end{bmatrix}; \tag{C.12}$$

Combining the Eq. C.4 and Eq. C.10 complete system dynamics will have form:

$$\xi_{full,k+1} = F x_{full,k} + G u_{c_2,k} \tag{C.13}$$

$$\xi_{full,k} = \begin{bmatrix} \xi_{s,k} & \xi_{c,k} \end{bmatrix} \tag{C.14}$$

$$F = \begin{bmatrix} F_{sys} & G_{sys} C_{cd} \\ G_{cd} C_{sys} & F_{cd} \end{bmatrix} \tag{C.15}$$

$$G = \begin{bmatrix} G_{sys} D_c \\ G_{cf} \end{bmatrix} \tag{C.16}$$

Bibliography

Abid, M. (2010), *Fault detection in nonlinear systems: an observer-based approach*, Ph.D. thesis, University of Duisburg-Essen.

Albu-Schäffer, A., S. Haddadin, C. Ott, A. Stemmer, T. Wimböck and G. Hirzinger (2007), The DLR lightweight robot: design and control concepts for robots in human environments, *Industrial Robot: An International Journal*, **vol. 34**, no.5, pp. 376–385.

Albu-Schaffer, A., C. Ott and G. Hirzinger (2007), A Unified Passivity-based Control Framework for Position, Torque and Impedance Control of Flexible Joint Robots, *The International Journal of Robotics Research*, **vol. 26**, no.1, pp. 23–39.

van Amerongen, J. and P. Breedveld (2003), Modelling of physical systems for the design and control of mechatronic systems, *Annual Reviews in Control*, **vol. 27**, no.1, pp. 87–117.

Aviezienis, A., J.-C. Laprie, B. Randell and C. Landwehr (2004), Basic Concepts and Taxonomy of Dependable and Secure Computing, *IEEE Transactions on Dependable and Secure Computing*, **vol. 1**, no.1, pp. 11–33.

Ball, R. (1900), *A Treatise on the theory of Screws*, Cambridge University Press.

Bäuml, B., T. Wimbock and G. Hirzinger (2010), Kinematically optimal catching a flying ball with a hand-arm-system, in *Intelligent Robots and Systems*, pp. 2592–2599.

Bemporad, A., W. P. M. H. Heemels and M. Johansson (2010), *Networked control systems*, Springer, Berlin/Heidelberg.

Bezemer, M. M. (2013), *Cyber-Physical Systems Software Development*, Ph.D. thesis, University of Twente.

Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2011), CSP-Capable Execution Framework, in *Communicating Process Architectures*, volume 68, pp. 157–175.

Bischoff, R., U. Huggenberger and E. Prassler (2011), KUKA youBot - a mobile manipulator for research and education, in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1–4.

Blanchard, B. S. and W. J. Fabrycky (2006), *Systems engineering and analysis*, Prentice Hall.

de Boer, H., T. S. Tadele and T. J. A. de Vries (2012), *Modeling and Control of the Philips Robot Arm*, Msc, University of Twente.

Bonitz, R. C. and T. C. Hsia (1996), Internal force-based impedance control for cooperating manipulators, *IEEE Journal of Robotics and Automation*, **vol. 12**, no.1, pp. 78–89.

Breedveld, P. C. (2004), Port-based modeling of mechatronic systems, *Mathematics and Computers in Simulation*, **vol. 66**, no.2-3, pp. 99–128.

Breedveld, P. C. (2008), Modeling and simulation of dynamic systems using bond graphs, in *Control Systems, Robotics and Automation, from Encyclopedia of Life Sup-*

port Systems, Eolss Publishers, Oxford, pp. 1–36.

BRICS (2011), BRICS Research camp 3.
`http://www.best-of-robotics.org/3rd_researchcamp/MainPage`

BRICS (2012a), BRICS - European Research Project - demonstration booth.
`http://www.automatica-munich.com/link/en/26335794#26335794`

BRICS (2012b), BRICS Research camp 5.
`http://www.best-of-robotics.org/5th_researchcamp/MainPage`

BRICS (2013), BRIDE - the BRIcs Development Environment.
`http://www.best-of-robotics.org/bride`

Brockett, R. W. (1984), Robotic manipulators and the product of exponentials formula, in *Mathematical Theory of Networks and Systems*, volume 58 of *Lecture Notes in Control and Information Sciences*, Ed. P. A. Fuhrmann, Springer Berlin Heidelberg, pp. 120–129, ISBN 978-3-540-13168-7.

Brodskiy, Y., D. Dresscher, S. Stramigioli, J. F. Broenink and C. Yalcin (2011), Design principles, implementation guidelines, evaluation criteria, and use case implementation for robust autonomy, Technical Report D61, The BRICS Project (Grant Agreement Number: 231940).
`http://www.best-of-robotics.org/`

Brodskiy, Y., R. Wilterdink, J. F. Broenink and S. Stramigioli (2013), Collection of methods for achieving robust autonomy, Technical report.

Broenink, J. F., J. F. Fitzgerald, C. J. Gamble, C. Ingram, A. H. Mader, J. Marincic, Y. Ni, K. G. Pierce and X. Zhang (2012), D2.3 — Methodological Guidelines 3, Technical report, The DESTECS Project (CNECT-ICT-248134).
`http://www.destecs.org/`

Broenink, J. F., M. A. Groothuis, P. M. Visser and M. M. Bezemer (2010), Model-Driven Robot-Software Design Using Template-Based Target Descriptions, in *ICRA 2010 Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications: How to modify and enhance commercial controllers*, IEEE, pp. 73–77.

Broenink, J. F., M. A. Groothuis, P. M. Visser and B. Orlic (2007), A model-driven approach to embedded control system implementation, *Control, Control*, pp. 137–144.

Broenink, J. F. and Y. Ni (2012), Model-driven robot-software design using integrated models and co-simulation, in *International Conference on Embedded Computer Systems,*, pp. 339 – 344.

Brugali, D., L. Gherardi, A. Luzzana and A. Zakharov (2012), A Reuse-Oriented Development Process for Component-Based Robotic Systems, in *Simulation, Modeling and Programming for Autonomous Robots*, pp. 361–374.

Brugali, D. and P. Scandurra (2009), Component-based Robotic Engineering Part I: Reusable building blocks, *Robotics Automation Magazine*, **vol. 16**, no.4, pp. 84–96.

Brugali, D. and A. Shakhimardanov (2010), Component-Based Robotic Engineering

(Part II): Systems and models, *Robotics Automation Magazine*, **vol. 17**, no.1, pp. 100–112.

Bruyninckx, H. (2001), Open Robot Control Software: the OROCOS project, in *Proceedings of IEEE International Conference on Robotics and Automation*, IEEE, pp. 2523–2528.

Bruyninckx, H., N. Hochgeschwender, M. Klotzbucher, P. Soetens, G. Kraetzschmar, D. Brugali, H. Garcia, A. Shakhimardanov, J. Paulus, M. Reckhaus, L. Gherardi and D. Faconti (2013), The BRICS Component Model: a Model-Based Development paradigm for complex robotics software systems, in *Proceedings of Annual ACM Symposium on Applied Computing*, ACM, 28, pp. 1758–1764.

Bruyninckx, H., P. Soetens and B. Koninckx (2003), The real-time motion control core of the Orocos project, in *Proceedings of IEEE International Conference on Robotics and Automation*, volume 2, Ieee, pp. 2766–2771.

Carlson, J. (2004), *Analysis of How Mobile Robots Fail in the Field*, Ph.D. thesis, University of South Florida.

Carlson, J., R. R. Murphy and A. Nelson (2004), Follow-up analysis of mobile robot failures, in *Proceedings of IEEE International Conference on Robotics and Automation*, volume 5, pp. 4987–4994.

Christophe, C., V. Cocquempot and B. Jiang (2004), Link between high-gain observer-based and parity space residuals for FDI, *Transactions of the Institute of Measurement and Control*, **vol. 26**, no.44, pp. 325–337.

Colgate, J. E. and N. Hogan (1987), Robust control of manipulator interactive behavior, *Modeling and Control of Robotic Manipulators and Manufacturing Processes, Modeling and Control of Robotic Manipulators and Manufacturing Processes*, **vol. 758**.

Colgate, J. E. and N. Hogan (1988), Robust control of dynamically interacting systems, *International Journal of Control*, **vol. 48**, no.1, pp. 65–88.

Colgate, J. E. and N. Hogan (1989), The interaction of robots with passive environments: Application to force feedback control, in *Fourth International Conference on Advanced Robotics*.

Conkur, E. S. and R. Buckingham (1997), Clarifying the definition of redundancy as used in robotics, *Robotica*, **vol. 15**, no.5, pp. 583–586.

Controllab Products B.V. (2013), 20-sim.
`http://www.20sim.com`

Czarnecki, K. and S. Helsen (2006), Feature-based survey of model transformation approaches, *IBM Systems Journal - Model-driven software development*, **vol. 45**, no.3, pp. 621–645.

Dassault Systemes AB (2013), Dymola.
`http://www.dymola.com`

De Santis, A., B. Siciliano, A. De Luca and A. Bicchi (2008), An atlas of physical human-robot interaction, *Mechanism and Machine Theory*, **vol. 43**, no.3, pp.

253–270.

Ding, S. (2008), *Model-based fault diagnosis techniques: design schemes, algorithms, and tools*, Springer.

Doty, K., C. Melchiorri and C. Bonivento (1993), A theory of generalized inverses applied to robotics, *Journal of Robotics Research*, **vol. 12**, no.1, pp. 1–19.

Dresscher, D., Y. Brodskiy, P. Breedveld and J. F. Broenink (2010), Modeling of the youBot in a serial link structure using twists and wrenches in a bond graph, in *Proceedings of SIMPAR 2010 Workshop, International COnference on Simulation, Modeling and Programming for Autonomus Robots, Darmstadt*, pp. 385–400.

Fassih, A., D. S. Naidu, S. Chiu and M. P. Schoen (2010), Precision grasping of a prosthetic hand based on virtual spring damper hypothesis, in *Proceedings of International Biomedical Engineering Conference*, pp. 79–82.

Franken, M. C. J. (2011), *Control of haptic interaction: An Energy-Based Approach*, Ph.D. thesis, University of Twente.

Franken, M. C. J. and S. Stramigioli (2011), Bilateral telemanipulation with time delays: A two-layer approach combining passivity and transparency, *IEEE Transactions on Robotics*, **vol. 27**, no.4, pp. 741–756.

Franken, M. C. J., S. Stramigioli, R. Reilink, C. Secchi and A. Macchelli (2009), Bridging the gap between passivity and transparency, in *Proceedings of Robotics, Science and Systems*.

Garcia, E. A. and P. Frank (1997), Deterministic nonlinear observer-based approaches to fault diagnosis: A survey, *Control Engineering Practice*, **vol. 5**, no.5, pp. 663–670.

Garg, V. and J. Hedrick (1995), Fault detection filters for a class of nonlinear systems, in *Proceedingd of the American Control Conference*, pp. 1647–1651.

Groom, K. N., A. A. Maciejewski and V. Balakrishnan (1999), Real-time failure-tolerant control of kinematically redundant manipulators, *IEEE Transactions on Robotics and Automation*, **vol. 15**, no.6, pp. 1109–1115.

Haddadin, S., A. Albu-sch, M. Frommberger and G. Hirzinger (2008a), The Role of the Robot Mass and Velocity in Physical Human-Robot Interaction - Part II : Constrained Blunt Impacts, in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1339–1345.

Haddadin, S., A. Albu-sch, A. D. Luca and G. Hirzinger (2008b), Collision Detection and Reaction : A Contribution to Safe Physical Human-Robot Interaction, in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 22–26.

Hochgeschwender, N., L. Gherardi, A. Shakhimardanov, G. Kraetzschmar, D. Brugali and H. Bruyninckx (2013), A Model-based Approach to Software Deployment in Robotics, in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE/RJS.

Hogan, N. (1985), Impedance Control: An Approach to Manipulation, *Journal of dynamic systems, measurement, and control*, **vol. 107**, no.March, pp. 1–24.

Hu, S. and W.-Y. Yan (2007), Stability robustness of networked control systems with respect to packet loss, *Automatica*, **vol. 43**, no.7, pp. 1243–1248.

Huntsberger, T. L., A. Trebi-Ollennu, H. Aghazarian, P. S. Schenker, P. Pirjanian and H. D. Nayar (2004), Distributed Control of Multi-Robot Systems Engaged in Tightly Coupled Tasks, *Autonomous Robots*, **vol. 17**, no.1, pp. 79–92.

Isermann, R. (1993), Fault Diagnosis of Machines via Parameter Estimation and Knowledge Processing Tutorial Paper, in *IFAC Symposium on Fault Detection, Supervision, and Safety for Technical Processes*, volume 29, pp. 815–835.

Isermann, R. (1997), Trends in the application of model-based fault detection and diagnosis of technical processes, *Control Engineering Practice*, **vol. 5**, no.5, pp. 709–719.

Isermann, R. (2006), *Fault-Diagnosis Systems An Introduction from Fault Detection to Fault Tolerance*, Springer-Verlag, Berlin/Heidelberg.

Kapsammer, E., T. Reiter and W. Schwinger (2006), Model-Based Tool Integration - State of the Art and Future Perspectives, in *Proceedings of International Conference on Cybernetics and Information Technologies*, p. 7.

Khatib, O., K. Yokoi and K. Chang (1996), Coordination and Decentralized Cooperation of Multiple Mobile Manipulators, *Journal of Robotic Systems*, **vol. 13**, no.11, pp. 755–764.

Klotzbucher, M., G. Biggs and H. Bruyninckx (2013), Pure Coordination using the Coordinator – Configurator Pattern, *CoRR*, *CoRR*, **vol. abs/1303.0**.

Klotzbucher, M. and H. Bruyninckx (2007), A Lightweight Real-Time Executable Finite State Machine Model for Coordination in Robotic Systems, Technical report. `http://www.wies-pro.de/pages/publications/KUL_RFSM-ecrts10.pdf`

Kraetzschmar, G., A. Shakhimardanov, J. Paulus, N. Hochgeschwender and M. Reckhaus (2010), Deliverable D-2.2: Specifications of Architectures, Modules, Modularity, and Interfaces for the BROCRE Software Platform and Robot Control Architecture Workbench, Technical report, BRICS FP7 project deliverable.

Kranenburg-de Lange, D. J. B. A. (2012), *Dutch Robotics Strategic Agenda - Analysis, Roadmap and Outlook*, June.

Laffranchi, M., N. G. Tsagarakis, D. G. Caldwell and W. B. Sheffield (2009), Safe human robot interaction via energy regulation control, in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, Ieee, pp. 35–41.

Locomotec (2010), KUKA youBot store. `http://www.youbot-store.com/`

Locomotec (2013), *KUKA youBot User Manual*.

Loncaric, J. (1987), Normal Forms of Stiffness and Compliance Matrices, *Robotics and Automation*, **vol. Ra-3**, no.6.

Lussier, B., A. Lampe, R. Chatila, J. Guiochet, F. Ingrand, M.-O. Killijian and D. Powell (2005), Fault Tolerance in Autonomous Systems: How and How Much?, in

Proceedings of IARP/IEEE-RAS/EURON Joint Workshop on Technical Challenge for Dependable Robots in Human Environments, pp. 1–10.

Mallet, A., C. Pasteur, M. Herrb, S. Lemaignan and F. Ingrand (2010), GenoM3: Building middleware-independent robotic components, in *Proceedings of IEEE International Conference on Robotics and Automation*, Ieee, pp. 4627–4632.

MathWorks (2013), MatLAB.
`www.mathworks.com`

Mellinger, D., M. Shomin, N. Michael and V. Kumar (2013), Cooperative Grasping and Transport Using Multiple Quadrotors, in *Distributed Autonomous Robotic Systems*, pp. 545–558.

Nakamura, Y. (1990), *Advanced robotics: redundancy and optimization*, Addison-Wesley Longman Publishing Co., Inc.

National Instruments (2013), LabView.

Niemeyer, G. (2004), Telemanipulation with Time Delays, *The International Journal of Robotics Research*, **vol. 23**, no.9, pp. 873–890.

Nomikos, P. and J. F. MacGregor (1994), Monitoring batch processes using multiway principal component analysis, *AIChE Journal*, **vol. 40**, no.8, pp. 1361–1375.

OMG (2013), The Object Management Group (OMG).
`http://www.omg.org/`

Open Source Open Modelica Consortium (2013), OpenModelica.
`https://www.openmodelica.org`

OpenRTM Project (2013), OpenRTM Project.
`http://www.openrtm.org/openrtm/`

Orocos Project (2013), Smarter control in robotics & automation.
`http://www.orocos.org`

Ott, C., B. Bäuml, C. Borst and G. Hirzinger (2005), Employing Cartesian impedance control for the opening of a door: A case study in mobile manipulation, in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems Workshop on Mobile Manipulators: Basic techniques, new trends and applications*.

Proteus Project (2013), Proteus.
`http://proteus.bourges.univ-orleans.fr`

Rajamani, R. (1998), Observers for Lipschitz Nonlinear systems, *IEEE Transactions on Automatic Control*, **vol. 43**, no.3, pp. 397–401.

Ryu, J., D. Kwon and B. Hannaford (2004), Stable Teleoperation With Time-Domain Passivity Control, *IEEE Transactions on Robotics and Automation*, **vol. 20**, no.2, pp. 365–373.

van der Schaft, A. (2000), *L2 - Gain and Passivity Techniques in Nonlinear Control*, Communications and Control Engineering, Springer, London.

Schlegel, C. (2013), SmartSoft: Components and toolchain for robotics.
`http://smart-robotics.sourceforge.net`

Schlegel, C. and R. Worz (1999), The software framework {SmartSoft} for

implementing sensorimotor systems, in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pp. 1610 –1616 vol.3.

Schmidt, D. C. (2006), Model-driven engineering, *Computer*, *Computer*, pp. 25–31.

Selig, J. M. (1996), *Geometrical methods in robotics, Monographs in Computer Sciences*, Springer Verlag, New York, first edition.

Siciliano, B. and O. Khatib (2008), *Springer handbook of robotics*, Springer Berlin Heidelberg.

Sözer, H. (2009), *Architecting fault-tolerant software systems*, Ph.D. thesis, University of Twente, Enschede, The Netherlands.

Stilwell, D. J. and B. E. Bishop (2000), Platoons of underwater vehicles, *Control Systems*, *Control Systems*, pp. 45–52.

Stramigioli, S. (1998), *From differentiable manifold to interactive robot control*, Ph.D. thesis, University of Delft.

Stramigioli, S. (1999), A passivity-based control scheme for robotic grasping and manipulation, in *Proceedings of IEEE Conference on Decision and Control*, December, pp. 2951–2956.

Stramigioli, S. (2001), *Modeling and IPC control of interactive mechanical systems: a coordinate-free approach*, volume 266, Springer, London.

Stramigioli, S. and H. Bruyninckx (2001), Geometry and Screw Theory for Robotics, in *Proceedings of IEEE International Conference on Robotics and Automation*.

Thau, F. E. (1973), Observing the state of non-linear dynamic systems, *International Journal of Control*, **vol. 17**, no.3, pp. 471–479.

USA Department Of Defense (1980), *Military Standard procedures for Performing a Faliure Mode, Effects and Criticality Analysis*, USA Department of defense.

USA Department Of Defense (1991), *Military Handbook reliability Prediction of Electronic Equipment*, USA Department of defense.

Venkatasubramanian, V., R. Rengaswamy, K. Yin and S. N. Kavuri (2003), A review of process fault detection and diagnosis. Part I: Quantitative model-based methods, *Computers & Chemical Engineering*, **vol. 27**, no.3, pp. 293–311.

Visinsky, M. L. (1991), Fault Detection and Fault Tolerance Methods for Robotics, Technical report.

Visinsky, M. L., J. R. Cavallaro and I. D. Walker (1994), Robotic fault detection and fault tolerance: A survey, *Reliability Engineering and System Safety*, **vol. 46**, no.2, pp. 139–158.

Welch, G. and G. Bishop (2006), An Introduction to the Kalman Filter, *In Practice, In Practice*, **vol. 7**, pp. 1–16.

White, J. and J. Speyer (1987), Detection filter design: Spectral theory and algorithms, *IEEE Transactions on Automatic Control*, **vol. Ac-32**, no.7, pp. 593–603.

Willems, J. (1972a), Dissipative Dynamical Systems Part II: Linear System with qaudratic Supply, *Archive for Rational Mechanics and Analysis, Archive for Rational Mechanics and Analysis*.

Willems, J. C. (1972b), Dissipative Dynamical Systems Part I: General Theory, *Archive for Rational Mechanics and Analysis, Archive for Rational Mechanics and Analysis*, **vol. 45**, pp. 321–351.

Willow Garage (2013), ROS project.
`http://www.ros.org`

Wilterdink, R. J. W., Y. Brodskiy and J. F. Broenink (2013a), Eclipse 20-sim update site.
`http://www.ce.utwente.nl/20sim/updates/`

Wilterdink, R. J. W., Y. Brodskiy, T. S. Tadele and J. F. Broenink (2013b), 20-Sim C-code generation templates and model-to-model transformations.
`https://git.ce.utwente.nl/20sim`

Wimbock, T., C. Ott, A. Albu-Schaffer and G. Hirzinger (2011), Comparison of object-level grasp controllers for dynamic dexterous manipulation, *The International Journal of Robotics Research*, **vol. 31**, no.1, pp. 3–23.

Wimbock, T., C. Ott and G. Hirzinger (2006), Passivity-based Object-Level Impedance Control for a Multifingered Hand, in *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4621–4627.

Acknowledgment

I can finally answer the question I have been asked every time I step into the lab: “Is it done, Yury?” Yes! I also would add: “It takes a village to write a book”. Knowing, that behind each book there are many people who helped an author to acquire results, polish the text or simply recuperate from writing in his free time, and experiencing it first hand feels very different. It is my greatest pleasure to thank every one who was next to me in this fours year long project and adventure.

First of all, I would like to thank my promoter prof. Stefano Stramigioli and my assistant promoter dr. ir. Jan Broenink for the opportunity to work in BRICS project and to do my research in the Robotic and Mechatronics group. Stefano, thank you for the inspiring new ideas and the extraordinary amount of freedom I was allowed in choosing my research topics and ways to run this project. Dear Jan, thank you for your help in writing the thesis, especially that you showed me how to see the differences between what is said in the text and what is meant to be said. I greatly appreciate your guidance from my first day in Twente in both scientific and organizational matters.

The work with BRICS team has been the most interesting and valuable experience. Many of the ideas described in this thesis are results from the discussion with members of BRICS team. I would like thank some member of the team separately. Alexey Zakharov thank you for the encouragement and support, your fascination with robotics and programming have been always a source of inspiration for me. Dear Robert thank you for all the help in running the project, implementing interesting demos and writing. Chapter 2 of this book would not be possible with out your help. Special thank to prof. Herman Bruyninckx, our discussions in the past years made many ideas this book much sharper, and your timely invitation to work at KULeven allowed me to finalize my Thesis.

Our path have crossed with Douwe when he came to me searching for a topic of his master thesis, since then we have been working together and learning from each other. Thank you for your never wavering desire to help and most interesting discussions on all sort of topics. I am extremely grateful for your help during the work on this book: in proofreading, reviewing and the translation of the samenvatting.

Douwe has jointed the group of the most amazing office-mates, I could have wished, completing the most “gezellig” office in the university. Dear Abeje, Bayan, Xiaochen, Bart, Oguzcan, Yunyun thank you for the board-game nights, coffee-breaks talks and scientific discussions. Leaving this everyday encouraging atmosphere was the saddest part of finishing my work at the Robotic and Mechatronics group. Abeje, special thanks to you for proofreading my texts, it was immensely helpful.

Jolanda and Carla deserve thanks for the help in organizing every day things. You have made my arrival in stay in Twente an easy journey. I also would like to thank all my colleges for the great working environment in the group, it was always nice to have lunches together and talk about everything.

Four years ago when I have agreed to reside in Macandra, I did not know that this building for students inspire horror in most people. Indeed, it can be quite depressing to stay in 16 square meter in a building that is not “quite clean”. Despite hideous architecture and fake fire alarms in the night, this building become a home for me form the moment I met Juan-Carlos, Tracje, Mehmet, Damla and Paula. Tracje thank you for sharing life-wisdom in evenings. Mehmet, thank you for playing guitar, it was real moments of peace. Juan-Carlos thank you for teaching me the Spanish-way and keeping me always informed about social life in Enschede. It was always nice to go climbing on weekends together and I will never forget the Dancing competition. I am grateful for your support.

There are many more people such as members of MTP group, Arashi and Rico Latino whom I meet over last four years, who have become good friends sharing good time together and helping in hard moments. I thank Bram, Basha, Bart, Lionel, Anika, Paul, Aysegul, Tjerik, Leo, Michael, Olga, Annutka, Wesley, Matteo, Marcel and all others for great time together.

Finally, I would like to thank my family for their never wavering support. My parents have never yet failed to plunge into a technical discussion giving suggestions which I almost always found valuable. To my grand parents, Natasha and Nicolay: Спасибо за вашу поддержку и советы. Мне всегда очень помогал ваш пример борьбы с невзгодами и умение наслаждаться жизнью.

This book is dedicated to my wife Edit. I am very grateful for the heavy snowfall of 2009, which stranded both of us in the middle of the Netherlands and allowed us to meet. It is due to Edit’s help and critics I was able to start and finish this book. Dear Edit köszönök mindent!!

2014, Leuven

About the author

Yury Brodskiy was born on September 15, 1984 in Leningrad, USSR. In 2007, he received his MSc. degree in Control Systems for Underwater Vehicles in Saint-Petersburg Marine Technical University, Russia. After working as technical consultant for Statoil in Norway, he has joined the research project Best Practice in Robotics in 2009 as a part of University of Twente team. In this project, he was responsible for research in the workpackage named “Robust Autonomy”. The goal of this research is to investigate and suggest ways to improve the reliability of a robot, with a focus on its motion control software. The results of four years of research are presented in this Thesis.

ISBN: 978-90-365-3620-2