Università degli Studi di Ferrara

DOTTORATO DI RICERCA IN
SCIENZE DELL'INGEGNERIA

CICLO XXVII

COORDINATORE Prof. Trillo Stefano

Design and Development of a Reusable Component-based Architecture for Surgical Robotics

Settore Scientifico Disciplinare ING/04

Dottorando
Dott. Preda Nicola

Tutore
Prof. Bonfe' Marcello

Anni 2012/2014

To my parents and Martina.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Marcello Bonfe', for the support during my Ph.D study and research.

Then, I would like to thank all the people who worked at the I-SUR project along the last three years: you all gave me the opportunity to grow both personally and professionally.

Finally I would like to thank my family and Martina, for being an important part of my life.

Abstract

Robotics is a growing field that is reaching a wide variety of application areas. The employment of robots for the implementation of a task is not anymore a prerogative of certain branches of the industry. In fact, more and more frequently robots are utilized to support humans during the execution of an assignment and this requires a flexible system, able to adapt to the environment.

Moreover, given the number of contexts in which robots are used, there is an increasing need for modular and reusable tools for the description of tasks. The robotic applications considered in this work are mainly related to robotic surgery, since minimally invasive surgery is a challenging field in which the employment of robots has enabled significant improvements in terms of quality of the procedures.

This study provides a set of patterns aimed to the design and the development of a component-based software architecture for the description of a complex robotic task. The best practices illustrated in this work are built on the concept of separation of concerns and have been defined to promote the creation of a reusable framework of components for the robotics. The proposed patterns are first introduced and then applied to different case studies to demonstrate their adaptability to describe a complex robotics task in different application domains.

Table of contents

Introduction

Robotics represents an expanding market and always more frequently robots are exploited for the automation of production processes. The advantages introduced by the use of a robot may vary from a reduction of the costs to an improvement in terms of quality of the process. In particular, in some contexts it is required a human-robot interaction and that means that the architecture controlling the robot must be able to adapt its behavior to the surrounding environment and to react to unexpected events: this is the case of robots used in surgery applications.

In recent years, in fact, the application of *Minimally Invasive Surgery* (MIS) in conjunction with robotics has brought significant improvements in terms of quality to many surgical procedures(see [1], [2], [3], [4], [5]). Anyway, the robots currently available on the marked are teleoperated devices (see [6] and [7]) lacking any kind of autonomy and therefore relying in terms of surgical performance exclusively on the perception and the dexterity of a human operator.

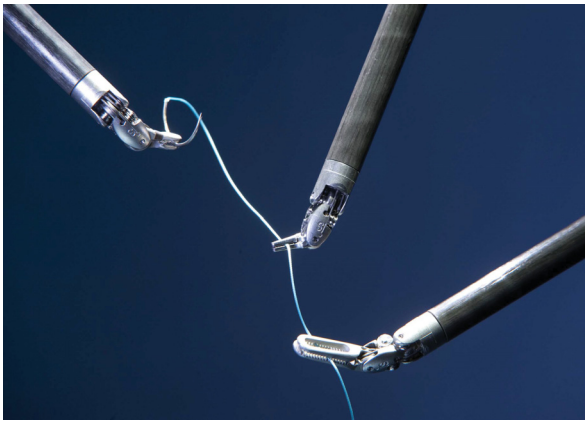

Fig. 1 Laparoscopic tools.

Even though automation has been exploited in several fields along the years, first of all the manufacturing field, the employment of autonomous robotics in the surgical context is a theme quite unexplored. The use of automation could, in fact, improve aspects as the safety,

the accuracy and the reproducibility of a task allowing, at the same time, to decrease the human fatigue ([8], [9]).

Fig. 2 DaVinci teleoperation.

To this end, the European Union funded a project called *Intelligent Surgical Robotics* (I-SUR) within the 7th Framework Programme. The project addresses the topic of the automation of surgical procedures and it aims to combine dexterity, sensing and cognitive capabilities to autonomously realize simple surgical tasks. Other than the improvements cited before, the introduction of an autonomous system would allow the surgeon to focus on the most difficult aspects of a procedures. The main objective of the project is, in fact, the implementation of elementary surgical actions such as the *puncturing* or the *suturing*.

Fig. 3 I-SUR project logo.

The author of this thesis has been involved in I-SUR, within the work package related to the implementation of the control part of the architecture but also the deployment, configuration and coordination of the system itself. The architecture has been developed following a component-based approach with the intention of generating modular and reusable components that could not only being exploited in the tasks required by I-SUR but also being employed for future works.

I-SUR Project Description

As anticipated, most of the patterns presented in this work have been developed in the context of the I-SUR project ([10]). The whole project can be split into seven work packages.

Model and Knowledge (WP1)

For the implementation of the I-SUR automatic robotic procedure it is necessary to understand how to represent the knowledge of a selected surgical action, and how to describe it with a form model. This work package is in charge of the modelling of the task: the models are defined through the interaction with the surgeons and the analysis of the surgical procedures.

Phantom and Organ Models (WP2)

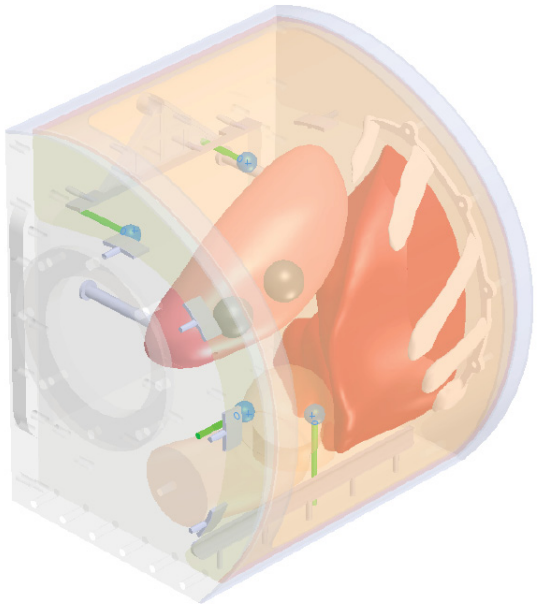

Fig. 4 Human abdomen phantom developed by WP2.

To verify the novel experimental surgical procedures phantom models have been developed (see 4) and present four advantages over animal or cadaveric models: biological safety, exemption from ethical review, low cost and long duration.

Sensing and Reasoning Module (WP3)

In order to enable a robotic system to automatically perform a procedure it is important that the system gets feedback from the environment and knows the current situation. The sensing

system uses many of the sense as the surgeon does and the reasoning module raises events if a critical situation occurs. This work package is in charge of the development and the implementation of tools for the integration of the sensing in the control architecture.

Robot (WP4)

This work package is in charge of the development of a robot prototype for the execution of the surgical tasks required by the I-SUR project. To do this it is required an interaction with WP1 and WP5 to define which are the constraints imposed by the considered surgical actions and how they can be respected from the point of view of the control.

Considering that the robot is placed near the patient, occupying space in an already crowded operating room environment, it needs to be as compact and lightweight as possible. At the same time, the robot has to be as rigid as possible to allow and accurate and fast positioning of the tools.

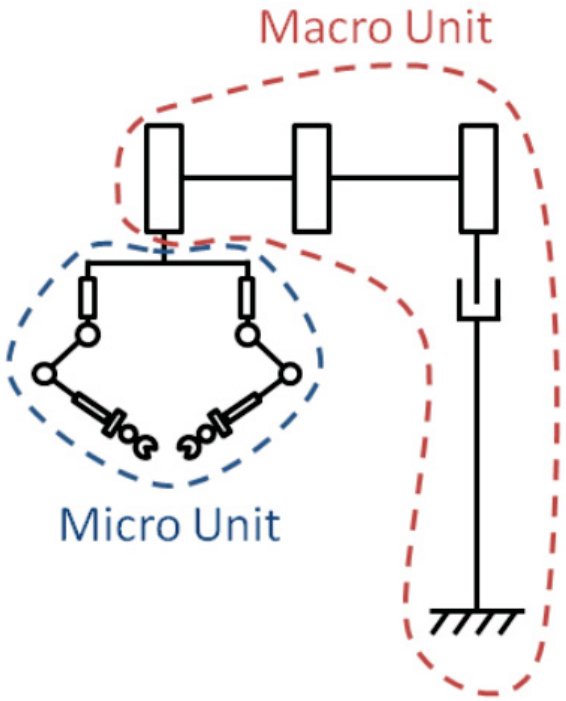

Fig. 5 Macro-micro kinematic concept.

To satisfy such constraints a *macro-micro* structure (see figure 5) has been chosen: the robotic platform is composed of a positioning manipulator (macro unit) and two redundant dexterous manipulators (micro units).

Control of the Surgical Actions (WP5)

This work package is in charge of the design and implementation of the control system. This is obtained starting from the models provided by WP1 and developing control strategies feasible for the robot projected by WP4. The control requirements can be seen as a sequence of modeling structures and control strategies that allow the reproduction of a surgical procedure.

The target is the implementation of autonomous surgical actions that can easily be followed and understood by the surgeon and in which a surgeon can take over control and continue the intervention manually if required.

Surgeon-robot Interface (WP6)

This work package is in charge of the development of a surgical interface with the objective of reducing the surgeon's surgical and cognitive load while improving safety and efficiency. The interface needs to communicate with the rest of the system to be able to provide a complete feedback about the execution of an underway surgical action.

Legal Aspects (WP7)

This work package is in charge of identifying the key aspects of the discipline of the surgical liability and the interaction between such liability and the utilization of new technologies in medical areas.

Research Objectives

During the I-SUR project it has been possible to extrapolate a set of generic design patterns aimed to the development of a component-based system. In particular, the focus of this work is to provide a set of design patterns for a component-based approach, aimed to increase the reusability of both the single components, when employed in different systems, and the whole architecture, when used to describe different tasks. For this purpose, guidelines will be provided about both the implementation of single components and the description of tasks.

Outline

In the first chapter, the essential concepts related to the component-based software engineering are introduced. The second chapter describes a set of patterns for the development of a reusable component-base software architecture for the robotics. The third chapter represents

and overview on the frameworks exploited by the author for the implementation of component-based systems and describes tools and libraries developed for this purpose. In the remaining chapters, it is provided a detailed description of three different case studies: for each case study, the required task is illustrated and then it is explained how the patterns described in the second chapter have been applied for the design and development of the related robotic architectures.

Chapter 1

Component-based Software Engineering

This chapter represents an introduction to the component-based software engineering. It contains a brief survey on the component-based software engineering and describes which are its key properties and requirements.

1.1 Background

The component-based development is a branch of software engineering that has as its main focus the *separation of concerns* of a software system: that separation is the key instrument for the implementation of loosely coupled components. The development of independent components is in favor of an approach that promotes the reuse of software elements. The goals of a reuse-based approach are primarily economic and in terms of the reduction of the cost, of the time and of the effort needed for the development of applications. Applications are in fact implemented deploying together prefabricated components and the design of loosely coupled components allow their employment across different contexts. The component-based approach was first introduced by Doug McIlroy in 1968 (see [11]) but over the time in was re-elaborated several times.

1.1.1 System Object Model (SOM)

The System Object Model is an IBM technology that enables languages to share class libraries regardless of the language they are written in. This ability of sharing class libraries between different object oriented languages helps to solve the reuse and interoperability problems between object oriented and not object oriented languages. SOM includes an interface definition language, a runtime environment with procedure calls and a set of enabling frameworks.

1.1.2 Component Object Model (COM)

The Component Object Model technology ([12]) is included in Microsoft Windows Operating Systems and enables software components to communicate. Is it used by developer for the creation of reusable software components, for their composition and for the interface with Windows services. The implementation of COM objects can be performed with several programming languages but the use of object-oriented languages, such as C++, makes it simpler.

1.1.3 Enterprise JavaBeans (EJB)

Enterprise JavaBeans is a platform for the creation of reusable, portable and scalable applications implemented in Java language. Each application in made of components contained inside an EJB container that provides them a set of services (related to security, transactions, web-services et al.). When a client application invokes a method on an EJB component, the call is passed through the EJB container first that performs these additional services and then passes the client's call to the EJB component. This process is transparent to the client application and permits to provide a variety of system services to the EJB components without the need of developing them every time.

1.1.4 Common Object Request Broken Architecture (CORBA)

Fig. 1.1 CORBA logo.

The Common Object Request Broken Architecture (CORBA, see [13]) is a standard defined by the Object Management Group (OMG) created to facilitate the communication of systems that are deployed on different platforms. CORBA has been designed to allow the collaboration between applications deployed on different platforms or written in different

programming languages. The main goals of this standard are reuse and encapsulation. CORBA uses an *Interface Definition Language* (IDL) to describe the interfaces that objects present to the outside and then provides a mapping between the IDL and the specific implementation language. The language mapping requires the developer to create IDL code that represents the interface of an object.

Since CORBA 3 the CORBA Component Model (CCM) has been added to the standard to provide an application framework for CORBA components. CCM is not language dependent as EJB but just like the Java platform it implements a component container where software components can be deployed. As for the EJB container the CCM component container provides a set of services to the components that it contains allowing to reduce the complexity of their implementation.

1.2 Component Definition

As suggested by the name itself, components are the fundamental bricks of a component-based architecture. To understand what a component is, it is better to start from the definition provided by the Object Management Group [14]:

> A component represents a *modular* part of a system that *encapsulates* its contents and whose manifestation is *replaceable* within its environment. A component defines its behavior in terms of provided and required *interfaces*. Larger pieces of a system's functionality may be assembled by *reusing* components as parts in an encompassing component or *assembly* of components, and *wiring* together their required and provided ports.

From this description some important properties of a component may be derived:

- **Modular.** A component is an independent element of the software that can be separated from the system and still maintain its specific functionality;
- **Encapsulated.** The internal state of the component should be unknown to the outside world;
- **Replaceable.** It should be possible to replace a component with an another one with a similar functionality with minor effort;
- **Reusable.** The design of a component should allow it to be reused in different scenarios; however a component can be designed for a specific task;

- **Connectable.** Two or more components can be connected through ports to allow *communication*;
- **Assemblable.** From the connection of components we obtain an assembly that can be seen as a component in itself. This is also called *composition*.

A software component can be implemented either with a *monolithic* implementation or with an *assembly*: in the first case the component is made of compiled code, in the second case in it obtained as a composition of other components. In both cases the result is a component that still needs to satisfy all the properties listed.

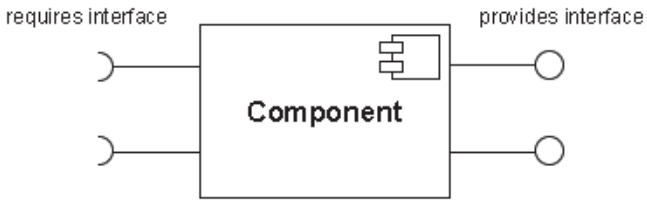

Fig. 1.2 UML representation of a component.

Package of Components

A package is a collection of components. There are different reasons to group some components. Sometimes it is useful to have interchangeable components absolving a pertinent function. It is the case, for example, of different kinds of controllers available inside a robotic system. Other times there may be the need of deploy the same component on different platforms and, for this purpose, it can be useful to create a package containing platform-dependent versions of the same component.

1.2.1 Interface

A component interface is a set of provided or required interfaces that are used to characterize its own behavior. First of all, an interface permits the connection of the component with other components through its *ports*. There can be input and output *ports* and they are used to implement a data-flow that allows the exchange of data between the components. Secondly, an interface can contain a set of properties that can be used to configure the component an thus affect the function the it provides. Finally, an interface can contain a set of *methods* that a component provides (or requires) to (or from) the rest of the system and that are a way to make it interact with the architecture other than the data-flow provided by its *ports*. As it will

be shown later, *methods* are particularly useful to create a connection between the reasoning and the computational sections of a system.

1.2.2 Deployment

Once a component has been compiled and combined into a package, it is ready to be deployed. The entity that deploys the components is called *deployer*. A deployer can execute the following operations:

- **Import.** A package containing different implementations is made available to the target environment;
- **Load.** A specific implementation of a component is instantiated;
- **Connect.** The instance is connected to other instances;
- **Configure.** A specific configuration is applied to the instance, different configurations are possible;
- **Start.** The instance is brought to an executing state;
- **Stop.** The execution of the instance is interrupted.

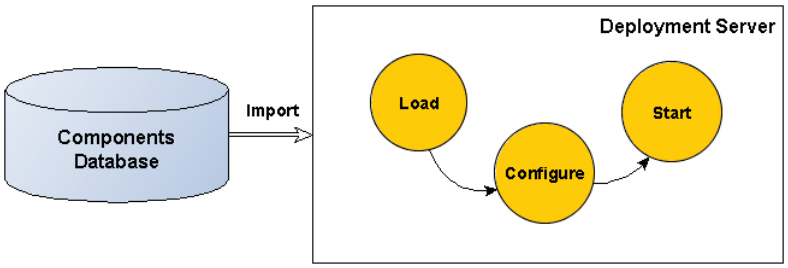

Fig. 1.3 Example of deployment phases.

The *deployment* is a fundamental part of a component based architecture because it is the mechanism that makes possible the process of *composition* and so it allows to obtain new components (i.e. new functions) starting from a set of available components. It follows that through the *deployment* the same component can be used in several applications promoting one of the key features of a component based architecture: the *reusability*. Unfortunately, not every component is reusable in different contexts and that often depends on the design choices taken for its implementation.

Chapter 2

Patterns for the Design of a Component-based Robotic Architecture

In this chapter a set of patterns is described: such patterns have proven to be helpful for the development of a complex component-based architecture for the robotics and they are the result of an insight mainly developed during the work at the EU-funded I-SUR project. These patterns have been built starting from the *5C's principle of separation of concerns* (see [15] and [16]) that separates *communication*, *computation*, *coordination*, *configuration* and *composition*. This approach is considered by the author a solid starting point for the development of modular, flexible and reusable architecture. On top of that, some guidelines are provided, related to the implementation of components and their deployment and coordination for the realization of a complex task.

2.1 Components and Roles

Before starting to talk about design concerns it is important to begin with the following quote [17]:

> "One thing can be stated with certainty: components are for composition."

Composition is in fact a key feature in every component-based architecture and this process is greatly facilitated if the components, the bricks of an assembly, are "well-designed". For sure a component may be considered "well-designed" when it satisfies the properties previously described in section 1.2, but it is not clear how to achieve such result.

When starting the design of a component a huge amount of choices are possible, depending on the final use of the object that is going to be created (e.g. the context in which the

component will be employed or the functionality that it should provide). The choices made by the designed at this phase will greatly influence the resulting component and that is why a set of patterns can seriously improve the development of a component.

From the author's experience, independently from the context in which an application is developed, it is possible to identify different types of components, depending on the function that they provide to the system and in relation with the *5C's principle of separation of concerns*. In particular, four components roles have been identified, respectively: *Calculation*, *Supervision*, *Decision* and *Bridge*. The characteristics of each set will be discussed later but, before that, it is important to underline a couple of properties shared by all these sets:

- grouping a set of components under a specific definition does not change their substance: they still are components (i.e. that means that they still need to satisfy all the properties previously listed);
- a component must be able to provide events or states to the rest of the system related to its own behavior (e.g. related to the completion of a function).

When developing a component it is suggested to think at the function that it should provide in the architecture and to check if this function fits into one of these families of components.

Often, designers are attracted by encapsulating many functions into just one component, but that generally leads to components that can be employed only in few specific deployments. It cannot be said that this is always a bad practice, because there are aspects, for example performance, that benefit from such approach (e.g. in the case of memory constraints), but from the point of view of the reusability this is a discouraged practice. The following sections give a description of each set of components in which their main properties are detailed.

2.1.1 Calculation Components

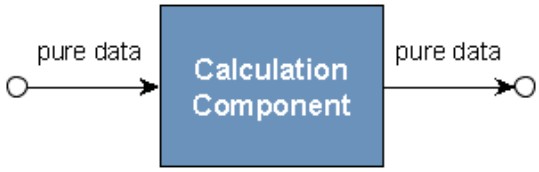

Fig. 2.1 Symbol for the *Calculation Component*.

A *Calculation* component is a pure *computation* component. This means that it is allowed to exchange only *pure data* (e.g., measures, numerical results) with the rest of the system. It generally encapsulates an algorithm that receives data through the input ports, processes it

and then send it to the system through the output ports. Allowing this family of components to handle *logic data* would make it dependent from a specific task, preventing it from be reused in different contexts without being modified. For example, internally modifying the behavior of an algorithm contained in a component on the base of a particular state of the current task forces the component to have an insight of the task itself. The suggested approach is instead to design a component that can be configured through its interface: in this way the reconfiguration is executed from the outside and no knowledge of the task is required from the inside.

2.1.2 Supervision Components

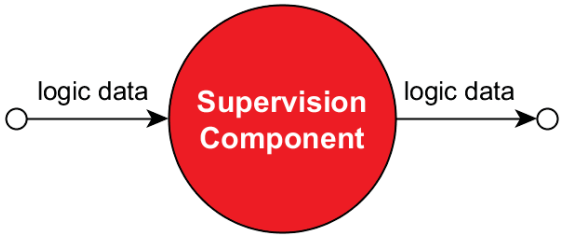

Fig. 2.2 Symbol for the *Supervision Component*.

A *Supervision* component is a pure *coordination* component. This means that it is allowed to exchange only *logic data* with the system in the form of events or states. It generally encapsulates an implementation of a behavioral tool (e.g., finite state machines, behavior trees et al.) that can receive events and states from the input ports and react to them; in output, events and states can be generated for the rest of the system. The fact that *pure data* is not allowed inside this family of components can be explained by the need of preserving the description of a behavior from the presence of numeric parameters that are generally dictated by a specific system. For example, if it is required to perform the same task with two different robots in two different contexts, leaving *pure data* outside this family of components would allow to reuse the description of the behavior in both cases because. Using this approach, all the system-dependent parameters are left outside the task description and can be delegated to the configuration of the system.

2.1.3 Decision Components

A *Decision* component is used to translate the results of the *computation* into meaningful information for the *coordination*. This means than, differently from the *Calculation* and the *Supervision* components, it is allowed to receive *pure data*, such as measures or numerical

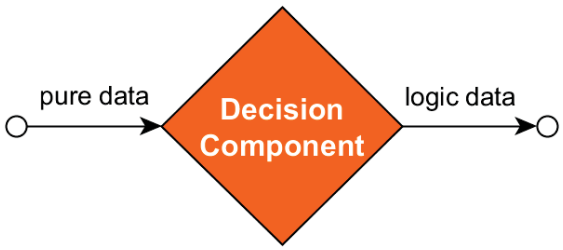

Fig. 2.3 Symbol for the *Decision Component*.

results available in the system, and to produce *logic data*, such as events and states for the supervisors of the system. To implement this passage from *pure data* to *logic data*, it generally encapsulates an implementation of a decision making mechanism (e.g., from a simple if-then implementation to more complex tools like Bayesian networks et al.). Representing a connection between *computation* and *coordination*, this component is generally strictly related to the task for which it has been developed.

2.1.4 Bridge Components

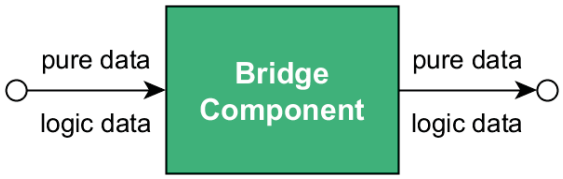

Fig. 2.4 Symbol for the *Bridge Component*.

A *Bridge* component is used to change the representation of an information so it does not strictly belong to *computation* or *coordination*: indeed a *Bridge* component can be considered as a support to the *communication*. The information, in fact, can be either in the form of *pure data* type or *logic data* type, what it matters is that the type of the data in input is preserved at the output. A *Bridge* component allows the *communication* between the system and an another entity that it is not directly represented in the system as a component. An example could be the introduction of a hardware device in the architecture that requires the use of a proprietary library: this could be encapsulated in a *Bridge* component that it would represent the device in the system and make it accessible by other components. An another example could be the the implementation of a communication between the architecture and a hardware device (or even an another system on a different platform) using a specific data transfer protocol (e.g. TCP/IP, UDP, EtherCAT et al.).

2.1.5 Knowledge of the Task

With reference to the roles of components just defined, a fundamental point is to define which ones are allowed to contain knowledge about a task. In the author's experience only two set of components should contain information about the final task of the architecture: *supervisor* and *decisor* components.

About the *supervisor* components, being them in charge of the description of the task through the coordination of the architecture, they encapsulate an intrinsic knowledge about the task.

For what concerns instead the *decisor* components, being them in charge of generating events and states for the coordination of the system, they are allowed to contain information about the task (i.e. how a set of data can be interpreted in the current task environment).

It is not possible to define an absolute rule about "how much" knowledge of a task must be injected in a component, and it would not make sense anyway because, in the end, this must be a choice of the designer. The suggestion given here is to always keep in mind that, in general, inserting some information about the task (e.g. states of the task) inside a component would hardly make it reusable without modifying its implementation.

Even for those components which function is strictly related to the advancement of the task (e.g. a component that provides set points to the system), it is still possible to achieve some results in terms in reusability. It has been said before that a *Decision* component must contain at least a partial knowledge of the task to be able to generate states and events for the reasoning part of the system starting from *pure data*. It could be the case, for example, of a wrench measured by a force sensor being over the maximum value allowed by the task: in this case it would be useful to expose the desired threshold value as a property of the component (through its interface) in order to make it configurable. Indeed the possibility of *re-configuring* a component must be kept in mind during its implementation, because it is a mechanism that provides flexibility and then reusability to the component and facilitates its employment in different tasks and architectures.

2.2 Task Description

The starting point in the design of a complex component-based architecture is the need of satisfying some requirements. Generally, a task can be decomposed into several sub-tasks: this process can be iterated on the sub-tasks until the level of complexity allows to provide the actions required by a state using primitives of actions available in the system. For example, a task like opening a door could be decomposed as shown in figure 2.5.

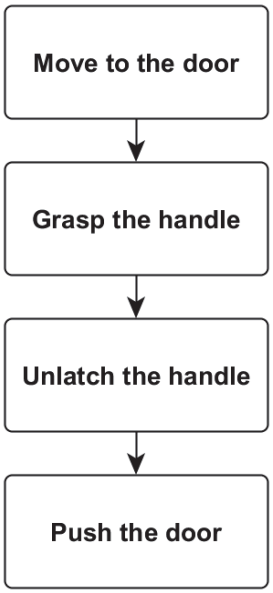

Fig. 2.5 State machine used to describe the "door task".

After this first decomposition it must be verified if these action are available as primitives of the system or if they must be further decomposed into simpler actions in order to be requested to the architecture.

2.2.1 Deployment

Which is the set of actions available in the system at a certain state is a matter of *deployment*. Through the deployment, in fact, it is possible to compose two or more components into an assembly of components. The actions made available by an assembly are not only function of the actions provided by the single components, but they are also a result of their peculiar composition. It is therefore possible to create an association between a *deployment* and a set of actions available to the architecture, and this specific *deployment* can then being associated to those states of the system in which such set of actions is required.

2.2.2 Configuration

It has been explained how, thanks to the deployment, it is possible to compose a set of components in order to make a specific set of actions available inside the system. Still, a deployment does not provide information about how these primitive actions will be performed: it is a matter of *configuration*.

Applying a *configuration* to the architecture requires to interact with its component through their interfaces in order to configure them with a precise set of parameters. For example, considering a component encapsulating the implementation of a mass-spring-damper system, applying a configuration to the component could mean changing the parameters of the virtual system and then its behavior.

2.2.3 Supervisors as Coordinators of the Architecture

To summarize, it has been stated that it is possible to provide a set of actions taking advantage of the *deployment*, and that the way in which these actions will be performed will be dependent on a certain *configuration*. Moreover, the task can be generally decomposed until the level at which the actions required by every single sub-task may be provided using primitive actions of the system (i.e. functions provided by components).

It follows that being able to implement a specific deployment and configuration, while being in a peculiar state of the task, allows to provide the actions required by such state. Therefore, each state of the task requires a precise couple of deployment and configuration.

In order to keep the task description clean from concerns regarding the deployment and the configuration of a certain architecture, it is suggested here to implement *composition* (i.e. deployment), *configuration* and *coordination* using three different *supervisor* components.

A similar approach has been proposed here [18], but in that case the coordinators where only two, a task coordinator and a configuration coordinator and the latter was in charge not only of the configuration of the components but also of their deployment (e.g. start and stop) and of the execution of actions. As represented in figure 2.6, a task can be described taking

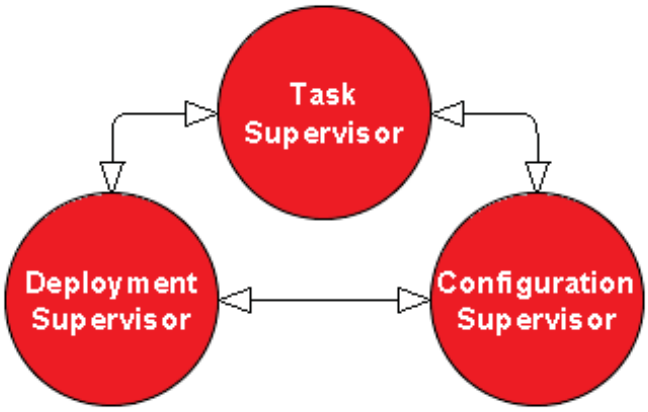

Fig. 2.6 Coordination between task, deployment and configuration supervisors.

advantage of the interaction and the coordination of three different supervisors.

The *task supervisor* is the one that should contain the full description of the application: for each state of the task this supervisor must coordinate with the *deployment supervisor* and the *configuration supervisor* accordingly with what required by the application in terms of actions and behaviors.

Having three different coordinators increases the flexibility of the system because the separation between deployment and configuration permits, for example, to maintain a set of actions (i.e. to maintain a certain deployment) changing the way in which such actions are performed (i.e. changing the configuration).

This kind of mechanism is particularly useful when the same task must be reproduced on a physical system different from the one on which the task have been developed. In fact, decoupling the configuration from the task facilitates the migration of the same application from a physical system to an another, since the configuration is generally linked to a real system on which the application has been tuned.

Chapter 3

Robotics Framework and Tools

This chapter is about the Open Robot Control Software (OROCOS [19]), the main framework used during the I-SUR project, and other tools used to support the implementation of the component-based architectures. There are alternatives to OROCOS: for example the Open Core Control software, based on OpenIGTlink [20], could have been a valid option, but was discarded for the absence of an extensive support to the implementation of control algorithms. Also the Robot Operating System (ROS [21]) was considered, but in this case the framework, lacking a hard real-time support, could not entirely fit the requirements of the project. On the contrary, OROCOS allows a hard real-time implementation and, at the same time, provides the tools for a smooth integration with the ROS framework and for the distribution of the system on different machines through the support to the Common Object Request Broker Architecture (CORBA [13]).

3.1 OROCOS

The Open Robot Control Software is composed of several parts: it follows an overview of the tools constituting the framework.

3.1.1 Toolchain

It is the core of the framework and implements tools for the description of components and for the support of real-time scheduling. Moreover, it supports the extension to other robotics framework (e.g. ROS [21], YARP [22]) and allows the setup, the distribution and the building of real-time software components. Besides, it takes care of the real-time execution and communication of software components. The Toolchain can be considered as a *middleware* because it sits between the operating system and the application.

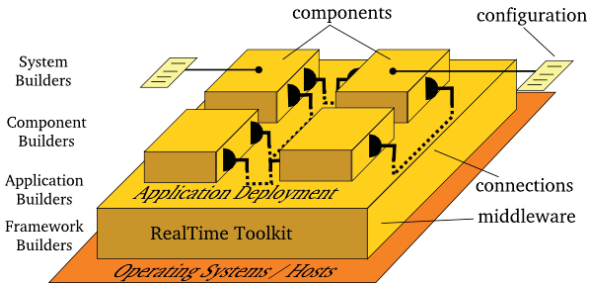

Fig. 3.1 Toolchain as middleware.

3.1.2 Kinematics and Dynamics Library (KDL)

It encapsulates a framework for the modelling and computation of kinematics structures. It includes a set of recursive solvers, for both kinematics and dynamics, and tools related to the generation of trajectories. It also implements a set of common geometric primitives of which it is provided a brief description:

- **Vector.** It describes a 3×1 matrix defined as follows:

$$Vector = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.1}$$

  Vectors support multiplication and division with a scalar, addition and subtraction with other vectors and cross and scalar products;

- **Rotation.** It describes a 3×3 matrix that represents a 3D orientation, it is defined as follows:

$$Rotation = \begin{bmatrix} X_x & X_y & X_z \\ Y_x & Y_y & Y_z \\ Z_x & Z_y & Z_z \end{bmatrix} \tag{3.2}$$

  A *Rotation.* object can be created in different ways in KDL:

  - as an identity matrix using *Identity()*;
  - from Roll-Pitch-Yaw angles using *RPY(roll,pitch,yaw)*;
  - from Euler Z-Y-Z angles using *EulerZYZ(alpha,beta,gamma)*;
  - built from Euler Z-Y-X angles using EulerZYX(alpha,beta,gamma);
  - from an equivalent axis-angle representation using *Rot(vector,angle)*.

  Moreover a *Rotation* object can be defined manually providing its elements, but in this case there are no controls on the consistency of the resulting orientation matrix. The

operations allowed on a *Rotation* object are: inversion, composition, multiplication with a *Vector*, and comparison.

- **Frame.** It describes a 4×4 matrix that represent a homogeneous translation matrix defined as follows:

$$Frame = \begin{bmatrix} X_x & X_y & X_z & x \\ Y_x & Y_y & Y_z & y \\ Z_x & Z_y & Z_z & z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} Rotation(3 \times 3) & Vector(3 \times 1) \\ 0(1 \times 3) & 1 \end{bmatrix} \tag{3.3}$$

  It is possible to construct a *Frame* starting from either a *Rotation* object, a *Vector* object or both. It is possible to calculate the inverse of a *Frame* and also its composition and comparison with other objects of the same type.

- **Twist.** It describes a 6×1 matrix and it is defined as follows:

$$Twist = \begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{3.4}$$

  It can be construct providing a *Vector* representing a translational velocity and a *Vector* representing an angular velocity. A *Twist* object supports the multiplication and division with a scalar and the addition, subtraction and comparison with other *Twist* objects.

- **Wrench.** It describes a 6×1 matrix and it is defined as follows:

$$Wrench = \begin{bmatrix} F_x \\ F_y \\ F_z \\ T_x \\ T_y \\ T_z \end{bmatrix} \tag{3.5}$$

It can be assembled providing a *Vector* representing a force and a *Vector* representing a torque. A *Wrench* object supports the multiplication and division with a scalar and the addition, subtraction and comparison with other *Wrench* objects.

It is important underline that using the $*$ operator between a *Vector* and a *Twist* or a *Wrench* allows to change their application point; similarly, using the $*$ operator between a *Rotation* and a *Twist* or a *Wrench* allows to change their reference frame. Using the same operator between a *Frame* and a *Twist* or a *Wrench* enables to change both their reference frame and their reference point at the same time.

3.1.3 Reduced Final State Machine (rFSM)

rFSM is an implementation of Statecharts (see [23]) that contains a subset (hence the name *reduced*) of the functions described by the UML specifications [24]. It is a standalone tool written in Lua [25] and this makes it portable and embeddable. In the context of OROCOS, it is mainly used for the coordination of a complex system but its function is not limited to that. It implements hierarchical states and allows the creation of state machines by composition of others state machines. Finally, it supports a real-time safe execution if Lua is configured to use the Two-Level Segregate Fit (TLSF [26]) as reported here [27]. It follows an example of Lua code implementing a simple finite state machine in the rFSM format, a graphical representation of the same finite state machine is shown in figure 3.2:

```
-- any rFSM is always contained in a state
return rfsm.state {

   on = rfsm.state {
      moving = rfsm.state {},
      waiting = rfsm.state {},

      -- define some transitions
      rfsm.trans{ src='initial', tgt='waiting' },
      rfsm.trans{ src='waiting', tgt='moving', events={ 'e_start' } },
      rfsm.trans{ src='moving', tgt='waiting', events={ 'e_stop' } },
   },

   error = rfsm.state {},
   fatal_error = rfsm.state {},

   rfsm.trans{ src='initial', tgt='on'},
   rfsm.trans{ src='on', tgt='error', events={ 'e_error' } },
   rfsm.trans{ src='error', tgt='on', events={ 'e_error_fixed' } },
   rfsm.trans{ src='error', tgt='fatal_error', events={ 'e_fatal_error' } },
   rfsm.trans{ src='fatal_error', tgt='initial', events={ 'e_reset' } },
}
```

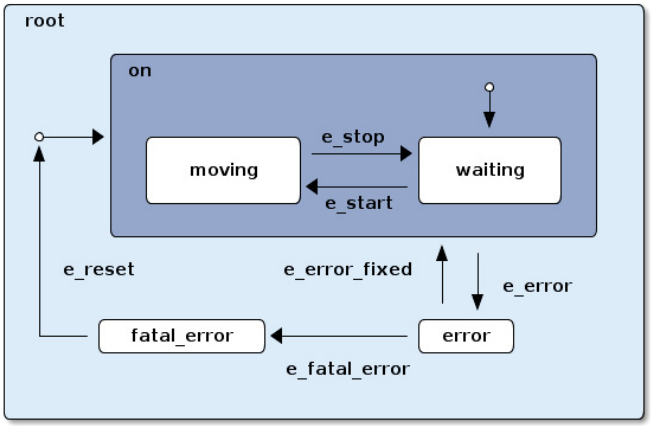

Fig. 3.2 Example of finite state machine.

3.1.4 Component

OROCOS components inherit from the *TaskContext* class that is defined in the Toolchain. Other than the C++ implementation contained in the Toolchain, it is alternatively possible to write a component in Lua using the RTT-Lua binding.

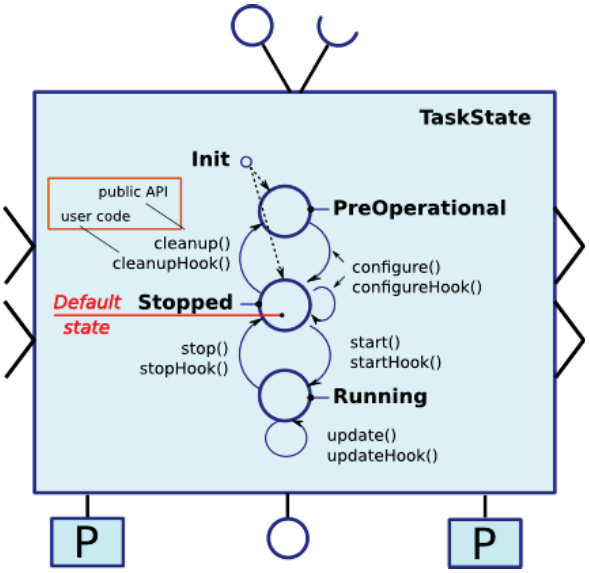

Fig. 3.3 TaskContext State Diagram.

Independently from the choice of the language used for the implementation, a component contains a set of *hook* functions that are related to its possible states, that are:

- **Init.** The initial state, just after the loading;

- **PreOperational.** The components has been created and it is ready to be configured;
- **Stopped.** The component is configured but not running.
- **Running.** The component is configured and is running accordingly with its *activity*.

A component must be loaded and then configured before being able to pass to the *running* state. While in *running* the behavior of a component depends on its activity: a component can be either synchronous or asynchronous, in the first case the Toolchain takes care of the scheduling of the component, otherwise a started component can be triggered sending data to a port configured as *EventPort*. That causes the thread of the TaskContext to trigger, executing one call of the method *UpdateHook()*.

3.1.5 Deployment

In OROCOS the deployment of an application is performed through the *deployer*. The *deployer* is an application based on the *DeploymentComponent* class and is responsible of creating applications starting from libraries of components. There are several versions of the *deployer* that can be used to run the deployment using a Lua engine, taking advantage of the CORBA transport or even under a Xenomai environment.

3.2 Additionally Developed Tools

Some additional tools have been developed by the author to support the design and the execution of component-based architectures within the OROCOS framework . It follows a brief description of how they work and how they can support the implementation of a complex application.

3.2.1 OroEdit

OroEdit is a tool written in Lua that has been developed to facilitate the design and the deployment of complex systems. From the interface of the program it is possible to create and edit components (represented as blocks) and to modify their interface. About the ports, it is possible to specify the following parameters:

- **Name.** The name of the port in the interface;
- **Port Type.** It specifies if the current object is an input or an output port; this field is not editable since its value is assigned at the creation, when the port type is selected;

- **Data Type.** It is possible to select the data type used by the port choosing it from a defined list containing all the most recurring *std* and *KDL* data types; moreover it is possible to specify a custom data type;
- **Event Port.** This field is available only for an input port and permits to select if the port must be defined as an event port.

For what concerns the properties, the following fields are available:

- **Name.** The name of the property in the interface;
- **Data Type.** Like for the ports, this field enables the selection of the data type associated with a property.

Moreover, it is possible to connect an input port with an output port, provided that they share the same data type. Indeed, after clicking on and input port and then on an output port their data types are checked and only if they match the connection is effectively created.

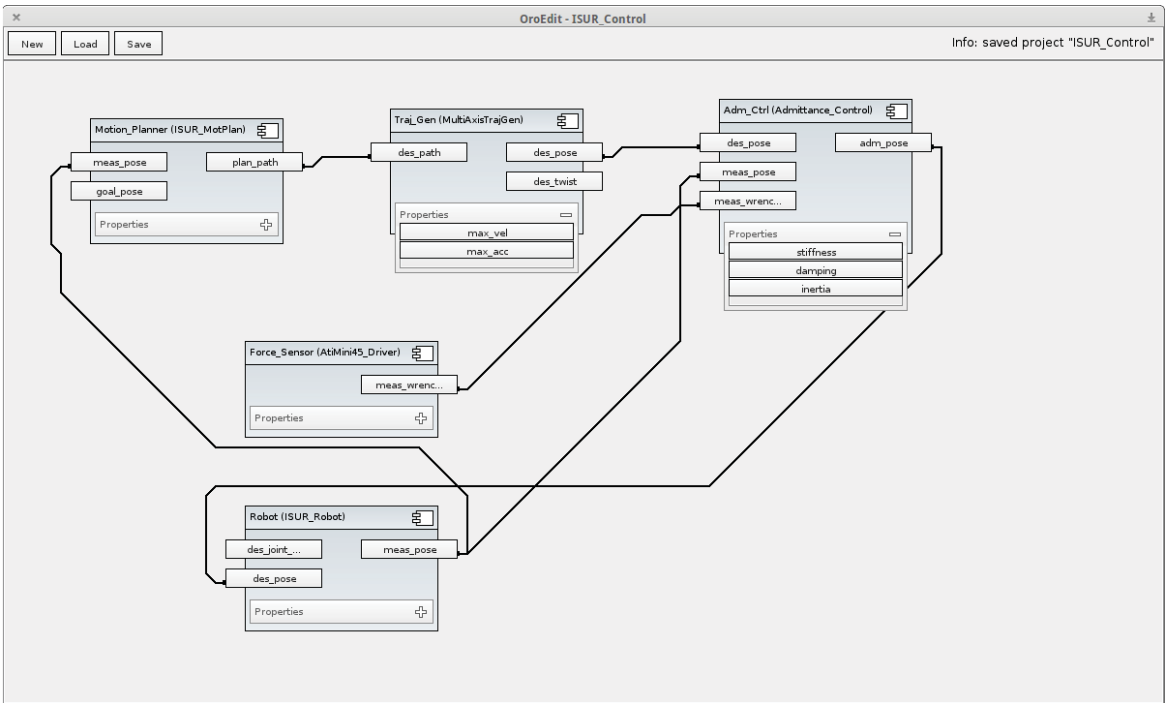

Fig. 3.4 OroEdit tool screenshot.

OroEdit permits to autonomously generate the code describing the interfaces of every component type included in the deployment. Moreover, it allows the generation of a simple deployment script that loads all the instances and connects them accordingly with the connections created in the tool. In figure 3.4 it is shown an example of architecture and it

follow the relative code, autonomously generated, for the constructor of the component type *Admittance Control*:

```
Admittance_Control::Admittance_Control(std::string const& name):TaskContext(name){

  //Ports
    this->ports()->addPort("des_pose", i_des_pose).doc("description");
    this->ports()->addPort("meas_pose",i_ meas_pose).doc("description");
    this->ports()->addPort("meas_wrench", i_meas_wrench).doc("description");
    this->ports()->addPort("adm_pose", o_adm_pose).doc("description");

  //Properties
    this->properties()->addProperty("stiffness", p_stiffness).doc("description");
    this->properties()->addProperty("damping", p_damping).doc("description");
    this->properties()->addProperty("inertia", p_inertia).doc("description");

}
```

Every project can be saved and loaded at will: all the instances, properties and connections are preserved and can be edited at any time.

An another feature that has been partially implemented at the time the author is writing is the possibility of exploring an existing deployment and to extrapolate a complete description of each instance and connection. This enables the creation of a library of components that can be used inside OroEdit for the description of other applications.

3.2.2 Configurator

Often, when an architecture reaches a high level of complexity, the maintenance of the deployment script may become difficult. Moreover, there may be the need of loading the same application with or without some optional functions and it can be hard to achieve such flexibility with a monolithic deployment script.

In order to increase the flexibility of the deployment and to make it easier to manage in combination with a supervisor, a *deployer configurator* module written in Lua has been developed. The *Configurator* module has been designed to replace a monolithic implementation of the deployment and it operates on Lua tables. For example, it is possible to request to the *deployer configurator* the start of an ordered list of components, or it is possible to set the activities of certain components accordingly with what specified in a table. The idea is that, working on tables, the management of the deployment can be performed just editing some configuration files, without modifying the deployment script itself.

Being able to describe the deployment just using tables (in this case Lua tables) enables also an easier interaction with an editing tool, such as OroEdit, that can just describe the deployment using configuration files instead of generating a whole deployment script.

Furthermore, leaving the parametric configurations of a component outside the deployment, accordingly with the pattern introduced in subsection 2.2.3, allows to separate the configurations of each instance, making them reusable in other applications without directly affecting the deployment script.

The *Configurator* provides a set of operations:

- **initConfigurator(***packages_list*, *components_list*, *configuration_folder_path***).** This function is used to initialize the *Configurator* and its use is mandatory. It requires a *packges_list* and a *components_list* that must contain, respectively, a list of the packages that must be imported and a list of the instances that must be created in the deployment. The *configuration_folder_path* argument must contain the path to a folder from which the configurations are loaded. It follows an example of *packages_list* and *components_list*:

```
packages_list = {
 "ocl",
 "kdl_typekit",
 "isur_motion_planner",
 "cartesian_traj_gen",
}

components_list = {
 { peer_name = "MotPlan", component_type = "ISURMotPlan"},
 { peer_name = "TrajGen", component_type = "CartesianTrajGen"},
 { peer_name = "TaskSup", component_type = "OCL::LuaComponent"},
 { peer_name = "Reporter", component_type = "OCL::FileReporting"},
}
```

- **configureComponents(***configuration_list***).** This function calls the configuration method of each component listed in *configuration_list* accordingly with their order. It can be called several time providing different lists. A list must be formatted as follows:

```
configuration_list = {
 "MotPlan",
 "TrajGen",
 "TaskSup"
}
```

- **startComponents(***start_list***).** This function calls the start method of each component listed in *start_list* accordingly with their order. It can be called several time providing different lists. A list must be formatted as follows:

```
start_list = {
 "MotPlan",
 "TrajGen",
 "TaskSup"
}
```

- **addPeers(***peers_list***).** For each entry of *peers_list* a set list of peers is added to one peer accordingly with this format:

```
peers_list = {
 { peer_name = "TaskSup", peers_to_add = { "Deployer, MotPlan" } },
 { peer_name = "Reporter", peers_to_add = { "MotPlan" } }
}
```

- **setActivities(***activities_list***).** For each entry of *activities_list* the activity of a component instance is set. The table *activities_list* must be formatted as follows:

```
activities_list = {
 {
  peer_name = "TrajGen",
  period = 0.001,
  priority = 95,
  scheduler = rtt.globals.ORO_SCHED_RT
 },

 {
  peer_name = "MotPlan",
  period = 0.01,
  priority = 0,
  scheduler = rtt.globals.ORO_SCHED_RT
 }
}
```

- **connectPorts(***connections_list***).** For each entry of *connections_list* a connection is created between two ports. The *Configurator* checks for the ports being respectively an *OutputPort* and an *InputPort* and then try to make the connection. The *connections_list* table must be written as follows:

```
connections_list = {
 { output = "MotPlan.path", input = "TrajGen.path", conn_policy = cp },
 { output = "MotPlan.events", input = "TaskSup.events", conn_policy = cp },
 { output = "TrajGen.events", input = "TaskSup.events", conn_policy = cp },
}
```

- **getReferences().** This function explores the current deployment and creates references for each component instance that is found. Each instance can be visualized from the console writing its name followed by parenthesis (e.g. *MotPlan()*). Moreover it is possible to access all the references for the ports, properties and operations of an instance interface writing the name of the instance followed by a dot and the name of the required element (e.g. *TrajGen.max_vel*).

3.2.3 Trajectory Generation Library (TGL)

The reason for which this library has been developed is the need of generating a multi-axis, multi-point trajectory with arbitrary initial positions, velocities and accelerations in the Cartesian space. Having to deal with a multi-arm system, which it is required to track a multidimensional planned path, it is necessary to ensure that each waypoint of a path is crossed by all the axes at the same time. To ensure that, it is possible to evaluate the slowest axis for each motion connecting two waypoints and to scale down the velocities of the other axes to make sure that the time employed for such motion is the same for each axis.

The algorithms implemented in the library can be found in [28]. A generic trajectory is represented by a base class *Trajectory* from which all the specific implementations inherit: the library includes polynomial trajectories until the seventh order. It follows a list of the main classes of the library:

- **Conditions.** It represents a set of constraints for the motion; initial and final *position* constraints are mandatory, all the other constraints, until the *jerk*, are optional and set to zero by default if not provided;

- **Trajectory.** It is the base class for all the implemented trajectories; at the construction a *Conditions* object must be provided, the duration of the trajectory is optional and set to one by default. Each trajectory type implemented in the library can be described as a polynomial function as follows:

$$q(t) = a_0 + a_1 t + a_2 t^2 + \cdots + a_n t^n \tag{3.6}$$

  where $t \in [t_0, t_1]$ with t_o and t_1 initial and final time instants, and the $n+1$ coefficients a_i are determined so that a set of initial and final constraints are satisfied (e.g. constraints on position, velocity, acceleration et al.).

- **MultiPointTrajectory.** It is the base class for the multi-point trajectories, the main difference is that instead than just the initial and final position, a whole path can be provided and all the intermediate velocities at the waypoints are automatically evaluated; in the library it has been implemented a *MultiPointLinearTrajBlends* class (that inherits from *MultiPointTrajectory*) that enables the generation of a multi-point linear trajectory with polynomial blends of order n.

  They are shown here the equations in the case of a linear trajectory with second order blends (i.e. parabolic blends), but the library enables the selection of a polynomial until the fifth order as blending function. In every case the trajectory is divided into three phases:

– **Acceleration phase.** In this phase $t \in [t_0, T_a]$ where T_a is the acceleration time. If $t_0 = 0$ for the second order case the trajectory is expressed as follows:

$$\begin{cases} q(t) = a_0 + a_1 t + a_2 t^2 \\ \dot{q}(t) = a_1 + 2a_2 t \\ \ddot{q}(t) = 2a_2 \end{cases} \tag{3.7}$$

where the coefficients a_i are determined so that the initial and final constraints are satisfied:

$$\begin{cases} a_0 = q_0 \\ a_1 = 0 \\ a_2 = \frac{v_v}{2T_a} \end{cases} \tag{3.8}$$

– **Constant velocity phase.** In this phase, if the acceleration and the deceleration phases have the same duration, $t \in [T_a, t_1 - T_a]$ and the trajectory is described as follow:

$$\begin{cases} q(t) = b_0 + b_1 t \\ \dot{q}(t) = b_1 \\ \ddot{q}(t) = 0 \end{cases} \tag{3.9}$$

and to ensure the continuity it derives:

$$\begin{cases} b_1 = v_v \\ b_0 = q_0 - \frac{v_v T_a}{2} \end{cases} \tag{3.10}$$

– **Deceleration phase.** In this phase $t \in [t_1 - T_a, t_1]$ and the trajectory is described as follow:

$$\begin{cases} q(t) = c_0 + c_1 t + c_2 t^2 \\ \dot{q}(t) = c_1 + 2c_2 t \\ \ddot{q}(t) = 2c_2 \end{cases} \tag{3.11}$$

with the c_i coefficients defined as:

$$\begin{cases} c_0 = q_1 - \frac{v_v t_1^2}{2T_a} \\ c_1 = \frac{v_v t_1}{T_a} \\ c_2 = -\frac{v_v}{2T_a} \end{cases} \tag{3.12}$$

The overall formulation is expressed as follows:

$$q(t) = \begin{cases} q_0 + \frac{v_v}{2T_a}(t-t_0)^2, & t_0 \leq t < t_0 + T_a \\ q_0 + v_v(t - t_0 - \frac{T_a}{2}), & t_0 + T_a \leq t < t_1 - T_a \\ q_1 - \frac{v_v}{2T_a}(t_1 - t)^2, & t_1 - T_a \leq t \leq t_1 \end{cases} \tag{3.13}$$

A similar formulation can be obtained in the case of polynomial blends until the fifth order.

- **MultiAxisConditions.** Similar to the *Conditions* class, it allows to specify constraints for multiple axes;
- **MultiAxisPath.** It is used to described a multi-point path for multiple axes;
- **MultiAxisTrajectory** This class enables to define a *MultiPointTrajectory* for an arbitrary number of axes: it allows to synchronize the trajectories along the whole path, scaling the velocities along each segment of the path accordingly with the duration of the motion of the slowest axis.

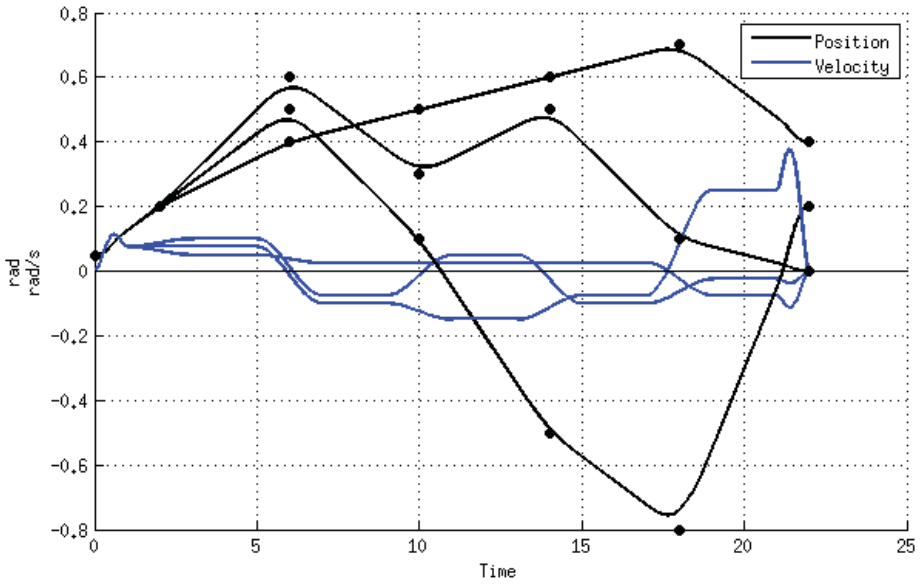

Fig. 3.5 Example of multi-axis, multi point trajectory.

In figure 3.5 it is possible to observe an example of use of the library for the generation of a multi-point trajectory in the case of three axes. In this case it has been generated a linear trajectory with polynomial blends of the fifth order. It may be observed that the waypoints are

not exactly crossed by the trajectory, with the exception of the initial and final positions. This is due to the need of linking two consecutive linear segments while respecting the velocity and acceleration constraints.

Chapter 4

Case Study: I-SUR *Puncturing* Task

In this chapter it will be introduced the puncturing task and it will be described in details the architecture developed for its implementation.

4.1 Puncturing Surgical Action

Puncturing is defined as the act of penetrating a biological tissue with a needle, e.g. to perform a biopsy or ablation techniques. The main goal of the puncturing is to reach a selected target point with a needle or a probe.

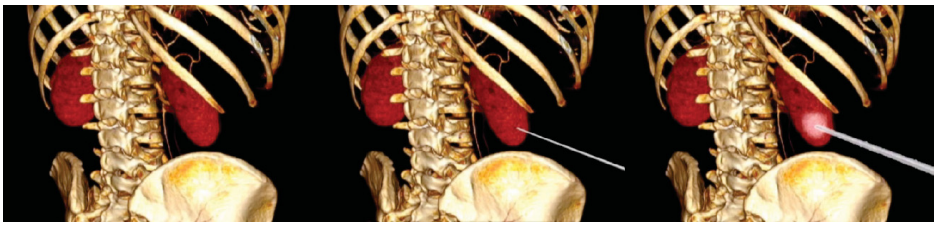

Fig. 4.1 Percutaneous cryoablation of posterior kidney tumors.

The surgical procedure analyzed in the I-SUR project is the *percutaneous cryoablation* of a small tumoral mass in the kidney (see figure 4.1) that is the least invasive treatment for kidney cancer. Percuraneous cryoablation requires the use of imaging devices (e.g., CT, MRI or Ultrasound) to precisely place one or more *cryoprobes* directly through the skin into the tumoral mass that needs to be destroyed. The cryoablation can be accomplished either during open surgery, laparoscopically or percutaneously (i.e. through the skin) with various modalities for image guidance. Within the I-SUR project it has been considered the percutaneous approach and an Ultrasound (US) probe has been used as guidance for the needles insertion.

4.2 Task Description

The main goal to be achieved is performing a puncturing procedure and it is decomposed into three sub-goals corresponding to the main phases of the task.

4.2.1 Planning

It involves the definition of the target zone to be reached and the identification of anatomical features of the zone, of the surrounding and of the forbidden regions that must be avoided (e.g., ribs, nerves, vessels, surrounding organs).

First, the information available from the image analysis are exploited in order to detect the cancer, the kidney and the forbidden regions.

Secondly, the number of needles used in the procedure must be defined and also their insertion points, their poses in relation to the tumor and their relative poses (to avoid collisions between the needles). All of this information are strictly connected since they are all contributing to the final coverage of the cancer.

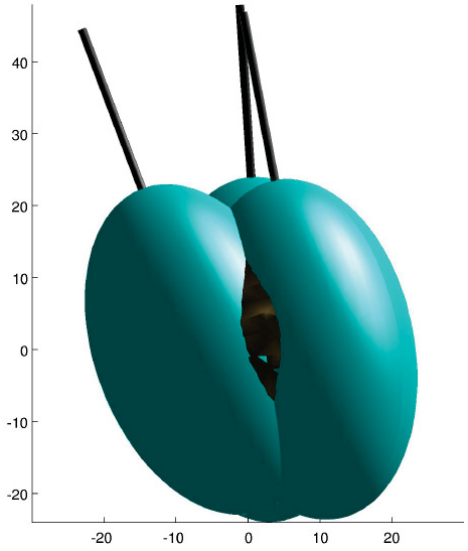

Fig. 4.2 Needle planning for a percutaneous cryoablation procedure

The planning is performed using a software tool [29] specifically developed for the project, that allows to consider a set of constraints and to determine the minimal number of needle and the poses required to obtain the best coverage of the tumor (see figure 4.2).

4.2.2 Puncturing Execution

In this phase, the needles must be positioned accordingly with the results of the planning: after every insertion the pose of the needle must be verified to guarantee that the procedure is proceeding correctly. From the point of view of the control, this phase can be decomposed into several parts.

First the US probe must be positioned in order to clearly see the tumor in the US images and to align the plane of the US probe to the direction of insertion of the needle. Otherwise the needle will not be visible in the US images and it will not be possible to verify if it has been correctly placed into the tumoral mass.

In a manually executed procedure the needle is tracked after its insertion, but in this case the pre-operative and the planning information can be exploited to predetermine an optimal arrangement of both the US probe and each one of the needles.

For this purpose, it has been developed an offline tool for the evaluation of an optimal placement of the US probe from the point of view of both the visibility of the tumor and of the needle. Given the surface of the skin, the position of the tumor and the placement of the needles provided by the cryoablation planner, it is evaluated a pose for the probe that improves the visibility of both needle and tumoral mass in the US images (see figure 4.3).

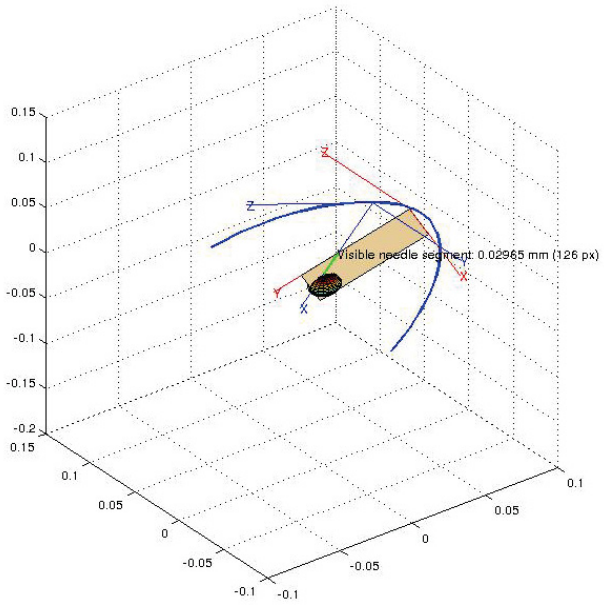

Fig. 4.3 Planning for the US probe.

Secondly the needle is moved until it reaches the surface of the skin. This is an important transition because before touching the skin the needle is in free motion, after that it is

in contact with the environment (i.e. the patient) and a different kind of interaction is required. Moreover, the human body is composed of several layers (e.g. epidermis, dermis, subcutaneous fatty tissue, et al.), each one describable with a different set of mechanical properties: the needle must be able to smoothly pass through these layers without applying forces that could damage the tissues.

Eventually the needle reaches the tumor and at this point its pose must be verified by the situation awareness module through the US imaging. Indeed the needle can blend during the insertion due to misalignments and friction in the needle-tissue interaction and this event must be intercepted by the system.

4.2.3 Needle Extraction

At the end of the cryoablation procedure, the needles need to be extracted. This operation can be potentially harmful for the tissues if the ice is not completely melted when the extraction begins: in this case the needle can be trapped by the ice and its extraction can cause damages to the tissues that generally involve bleeding.

To avoid this situation the force on the needle in monitored during the extraction and, as soon as it reaches a threshold value, the procedure is stopped: at this point the surgeon can decide to both wait for the ice to melt or to take over control and continue the intervention manually.

4.3 Hardware Setup

At the time of the experiments relative to the puncturing task, the prototype of the I-SUR robot only mounted one micro unit, used to hold the needle. Therefore, a second robot, a UR5 form Universal Robots, has been employed to hold the US probe (see figure 4.4).

4.3.1 I-SUR Robot

As anticipated, the I-SUR robot was designed following a macro-micro concept:

- **Macro-Unit.** The parallel robot used for the macro-unit is a linear Delta robot ([30]). The robot has three actuated prismatic joints, each one actuated by a geared DC motor connected to a ball-screw drive to generate translatory motion. Each of the moving parts of the ball-screw drives is then connected to a pair of parallel rods through a pair of passive universal joints. Finally, another pair of passive universal joints connects the other ends of the rods to the one common moving platform.

Fig. 4.4 Experimental setup for the puncturing task.

- **Micro-Unit.** By design, each micro-unit is composed of 7 degrees of freedom (DOFs). However, for the puncturing task only one arm was employed and this arm only had 4 DOFs. It has been used to hold the needle.

For the puncturing setup the I-SUR robot had a total of 8 DOFs, 4 DOFs provided by the macro structure (three prismatic joints plus one revolute joint in the base of the parallel structure) and 4 DOFs provided by the micro structure (all revolute joints).

The technical specification of the robot are the following:

Macro Payload	5 kg / 11 lbs
Macro DoF	4
Micro DoF	4 (7 in the final version)
Communication	UDP

4.3.2 UR5 Robot

The UR5 is an industrial robot produced by Universal Robots with a serial structure composed of 6 DOFs (see figure 4.5). It has been utilized to hold a commercial Ultrasound probe using an ad hoc adapter.

The technical specification of the robot are the following:

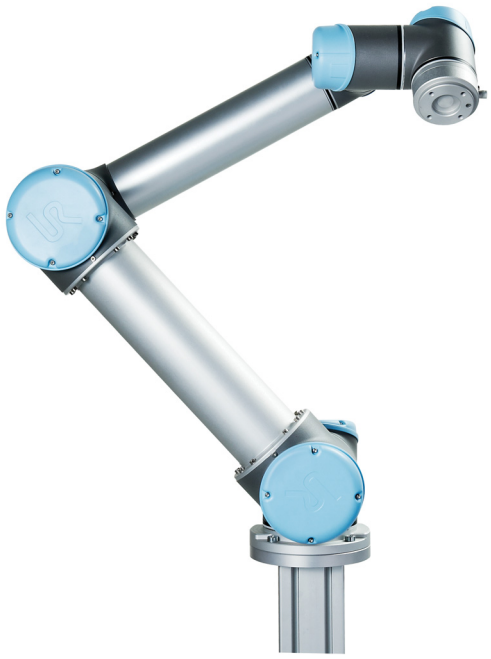

Fig. 4.5 UR5 robot.

Weight	18.4 kg / 40.6 lbs
Payload	5 kg / 11 lbs
Reach	850 mm / 33.5 in
Joint ranges	+/- 360° on all joints
Speed	Joint: Max 180°/sec. Tool: Approx. 1 m/sec./Approx. 39.4 in/sec.
Repeatability	+/- 0.1 mm / +/- 0.004 in (4 mil)
Degrees of freedom	6
Communication	TCPIP - Ethernet sockets, Modbus TCP

4.3.3 AtiNano17 Sensor

The AtiNano17 (figure 4.6) is a force and torque sensor (produced by ATI Industrial Automation) mounted on the wrist of the micro-unit of the I-SUR robot. The sensor has 6DOFs and it is used to measure the interaction forces between the arm and the environment.

The specifications of the sensor are listed as follows:

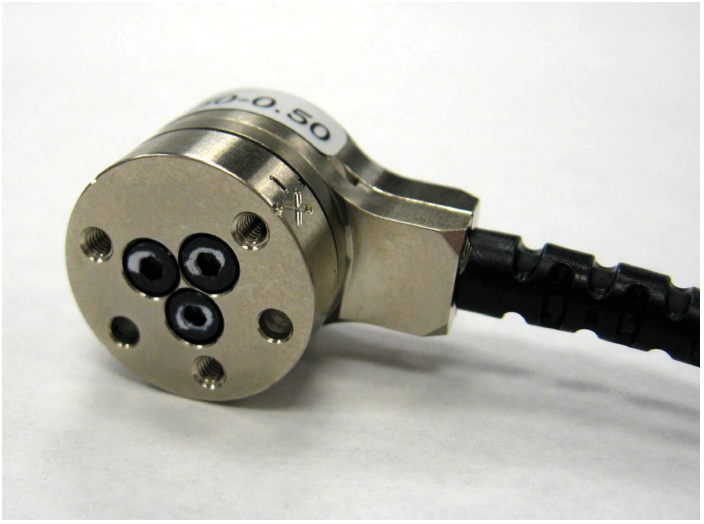

Fig. 4.6 AtiNano17 force and torque sensor.

Weight	0.00907 kg
Diameter	17 mm
Height	14.5 mm
Overload Fxy	±250 N
Overload Fz	±480 N
Overload Txy	±1.6 Nm
Overload Tz	±1.8 Nm
Stiffness X-axis & Y-axis forces (Kx, Ky)	8.2×10^6 N/m
Stiffness Z-axis force (Kz)	1.1×10^7 N/m
Stiffness X-axis & Y-axis torque (Ktx, Kty)	2.4×10^2 Nm/rad
Stiffness Z-axis torque (Ktz)	3.8×10^2 Nm/rad
Resonant Frequency Fx, Fy, Tz	7200 Hz
Resonant Frequency Fz, Tx, Ty	7200 Hz

4.3.4 PhantomOmni

During the project it has been used a Phantom Omni (see figure 4.7), a haptic device produced by Sensable with an IEEE-1394a FireWire interface. The latest version of the device is instead produced by Geomagic and has an Ethernet interface. The Phantom Omni has 6 DOFs: the position of each joint is measured but only the first three joints, the ones used for the translatory motion of the device, are actuated.

The technical specification of the haptic device are the following:

Weight	3 lb 15 oz
Force feedback workspace	160x120x70 mm / 6.4x4.8x2.8 in
Nominal position resolution	0.055 mm / 450 dpi
Backdrive friction	< 0.26 N / 1 oz
Maximum exertable force at nominal position	3.3 N / 0.75 lbf.
Continuous exertable force	> 0.88 N / 0.2 lbf.
Stiffness X axis	> 1.26 N/mm / 7.3 lb/in
Stiffness Y axis	> 2.31 N/mm / 13.4 lb/in
Stiffness Z axis	> 1.02 N/mm / 5.9 lb/in
Inertia (apparent mass at tip)	45 g / 0.101 lbm.
Communication	IEEE-1394a FireWire

Fig. 4.7 Phantom Omni haptic devicer.

4.4 Task Formalization

The puncturing procedure described at the beginning of this section has been formalized using finite state machines implemented with the support of rFSM (see subsection 3.1.3). The puncturing task requires a coordination between the robot holding the needle and the robot holding the US probe; moreover, in the hardware setup previously shown each robot controller is physically distributed on a different computer platform.

For this reason, it has been chosen to split the puncturing task into two sub-tasks, each associated to a robot: the insertion of a needle, executed by the I-SUR robot, and the positioning of a US probe, executed by the UR5 root. In figures 4.8 and 4.9 it is possible to observe the two finite state machines (FSM) describing the two tasks.

4.4.1 Needle Insertion FSM

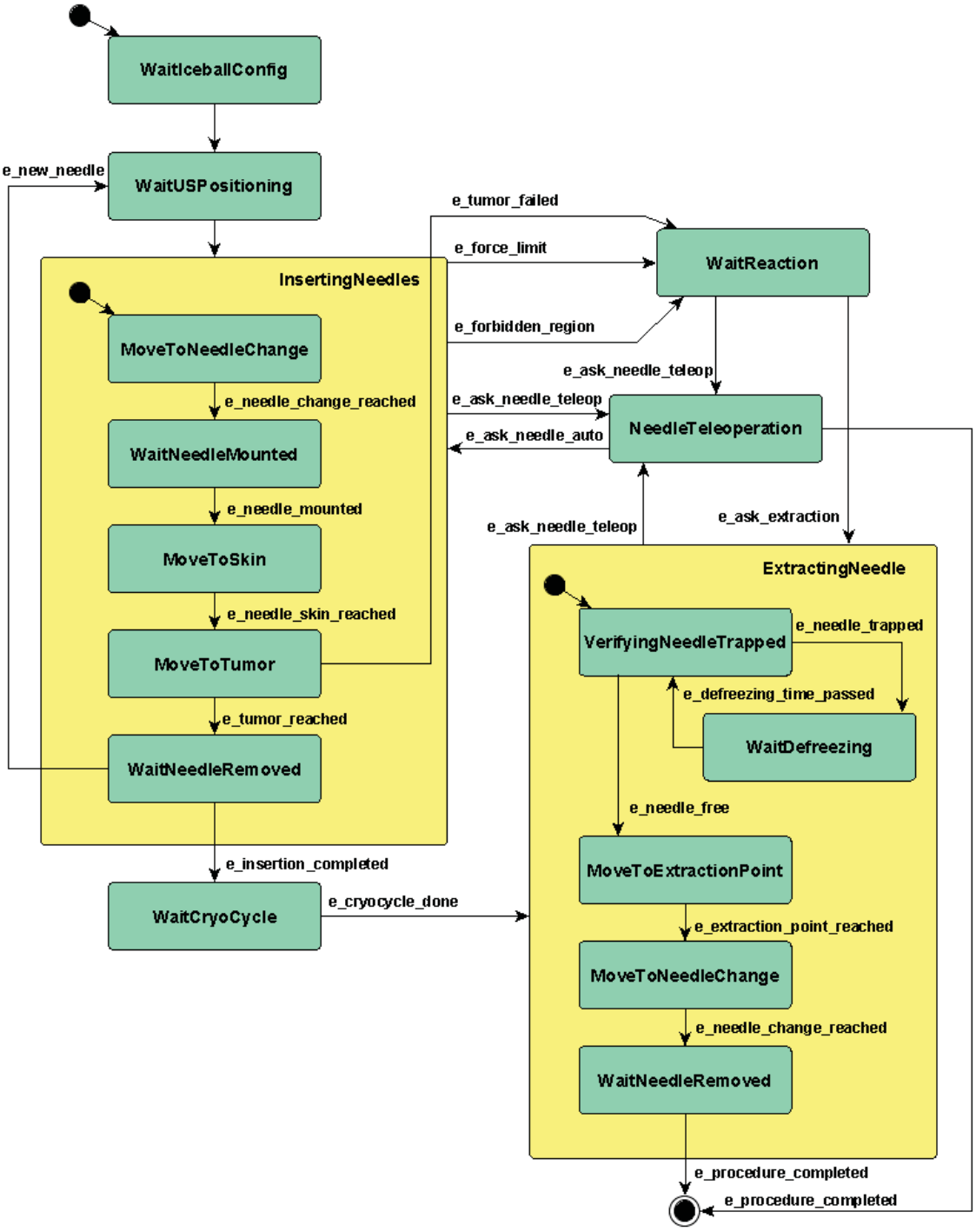

Fig. 4.8 Finite state machine for the needle insertion.

It follows a brief description of the states of the needle insertion task:

- **WaitIceballConfig.** The system waits for the iceballs planning completion;
- **WaitUSPositioning.** The needle robot waits for the US being in contact with the skin and with the right orientation;
- **InsertingNeedles.** In this macro-state a full needle insertion is described;
- **MoveToNeedleChange.** A free-motion to a predefined pose in which a new needle can be mounted on the end effector;
- **WaitNeedleMounted.** The system waits for a confirmation coming from the surgeon interface that a new needle has been mounted;
- **MoveToSkin.** A free-motion that ends when the needle touches the surface of the skin: this event is detected by the situation awareness;
- **PenetratingSkin.** A motion primitive that ends when the needle penetrates the surface of the skin: this event is detected by the situation awareness;
- **MoveToTumor.** In this phase the needle is penetrating the body of the patient; unexpected events like touching a forbidden region or applying a force over the threshold value cause a transition to the *Reaction* state. The same happens if, at the end of this motion, the tip of the needle fails to reach the target point on the tumor;
- **WaitNeedleRemoved.** The needle is manually detached from the robot and the system waits for a confirm that this action has been completed;
- **WaitCryoCycle.** During this phase the cryoablation procedure takes place: the task can advance after a confirm from the surgeon;
- **WaitReaction.** Every time that an unexpected event arises, the system make a transition to this state. While in this state the surgeon can decide how to react to the current event: the choice is between let the system recover from the failure or take over control and switch to the teleoperated mode;
- **NeedleTeleoperation.** While in this state the surgeon can directly control the robot through a haptic device;
- **ExtractingNeedle.** This macro-state encapsulate the procedure for a complete needle extraction;

- **VerifyingNeedleTrapped.** As stated before, after a cryoablation procedure the tip of the needle can be trapped by the ice and while in this state its extraction is particularly dangerous and can cause bleeding. For this reason, before starting the motion of extraction the situation awareness, using the measured wrench on the needle, estimates if the needle is free from the ice;
- **WaitDefreezing.** If the needle is trapped by the ice the only solution is to wait until the iceball on the tip of the needle defrosts;
- **MoveToExtractionPoint.** This is the motion that executes the extraction of the needle.

4.4.2 US Positioning FSM

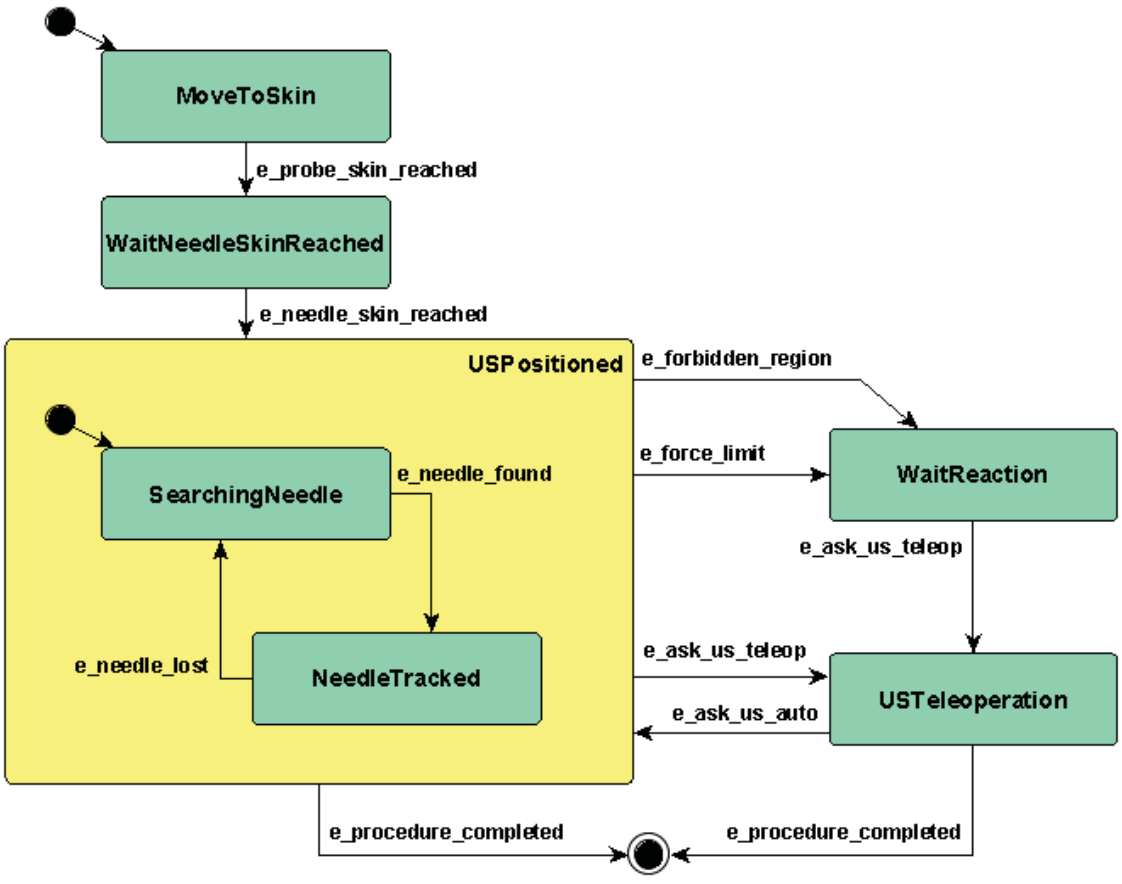

Fig. 4.9 Finite state machine for the us positioning.

- **MoveToSkin.** A free-motion that ends when the probe touches the surface of the skin;

- **WaitNeedleSkinReached** The US robot waits until the robot holding the needle reaches the surface of the skin;
- **USPositioned** While in this macro-state the probe applies a constant force to the surface of the skin in order to guarantee the visibility of the US image;
- **Searching Needle** In this state the orientation of the probe is changed following predetermined motion patters while the probe remains in contact with the skin; this process continues until both the tumor and the needle are visible in the US imaging;
- **NeedleTracked** The task stays in this state until the needle is detected in the US images by the situation awareness; otherwise a transition is performed toward the *Search* state;
- **WaitReaction.** Every time that an unexpected events arises the system make a transition to this state. While in this state, the surgeon can decide how to react to the current event: the choice is between let the system recover from the failure or take over control and switch to the teleoperated mode;
- **USTeleoperation.** While in this state, the surgeon can directly control the robot through a haptic device; this can be useful in the case of the probe when the system loses the needle tracking in the US images and is not able to autonomously recover from this state;

4.5 Developed Components

For what concerns the control part of the architecture, several components have been developed to satisfy the motion constraints imposed by a surgical task. In particular, two are the requirements considered here related to the motion of the system:

- collisions must be avoided in every case, the only contacts allowed are those required by the task (i.e. the needle interacting with the tissues);
- the motion of the robot must be compliant, unless otherwise requested by the task (e.g. during the penetration of the skin the behavior of the robot must be stiff, at least along the direction of the puncturing).

The following sections provide a detailed description of the components developed for the control part of the architecture.

4.5.1 Motion Planner

The motion planner is a *Calculation* component that searches a valid path between two given start and goal states (respectively q_{start} and q_{goal}). For the proposed setup it has been made the choice of describing the task in the Cartesian space; moreover, in anticipation of working with the definitive setup including the complete macro-micro robotic structure developed for the project, it has been chosen to simultaneously plan a path for both the arms. That means that the motion planner works on samples $q \in SE(3) \times SE(3)$. For the planning it has been exploited the Open Motion Planning Library (OMPL, see [31]) that consists of a collection of sampling-based motion planning algorithms. OMPL itself does not contain any code related to the collision detection and for this purpose it has been choosen the Flexible Collision Library (FCL, see [32]).

- **Open Motion Planning Library.** The representation of a sample $q \in SE(3) \times SE(3)$ has been defined as a *CompoundStateSpace* object. This OMPL class allows users to create arbitrarily complex state spaces out of simpler state spaces: in this case it has been used to compose two $SE(3)$ spaces. The path search algorithm employed is the *Rapidly-Exploring Random Trees Connect* (see [33]) that is a variant of the *RRT* algorithm.

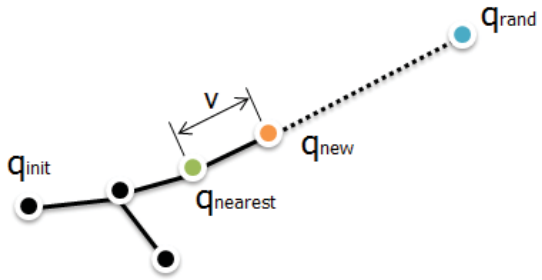

Fig. 4.10 RRT algorithm.

The basic RRT algorithm (see figure 4.10), attempts at each iteration to extend a tree by adding a new vertex q_{rand} that is biased by a randomly-selected configuration. At this point the nearest vertex in the three to the sampled configuration, $q_{nearest}$, is selected. A motion from the vertex $q_{nearest}$ to the sample q_{rand} is performed with a fixed incremental distance v and the resulting vertex q_{new} is tested against collisions. If q_{new} is a valid sample, it is added to the tree (or either q_{rand} may be added if it is within v). The use of an incremental motion is mainly due to the need of solving path planning problems that involve differential constraints.

The RRT-Connect algorithm is instead designed specifically for path planning problems that do not involve differential constraints. In particular the method is based on two ideas:

- the *connect* heuristic that attempts to move over a longer distance;
- the growth of RRTs from both q_{start} and q_{goal}.

In the case of the RRT-Connect, once a new vertex q_{rand} has been sampled, the *connect* heuristic function does not just try to extend the current RRT by a single v step, but iterates the extension steps until a collision is detected or q_{rand} is reached.

- **Flexible Collision Library.** FCL is a library that permits to perform different types of proximity queries on a pair of geometric models composed of triangles (see figure 4.11). The library allows the collision detection between two models: it can handle the case of two moving models and it can optionally provide information about the contact (e.g. contact normals and contact points). Moreover it enables the computation of the minimum distance between two models.

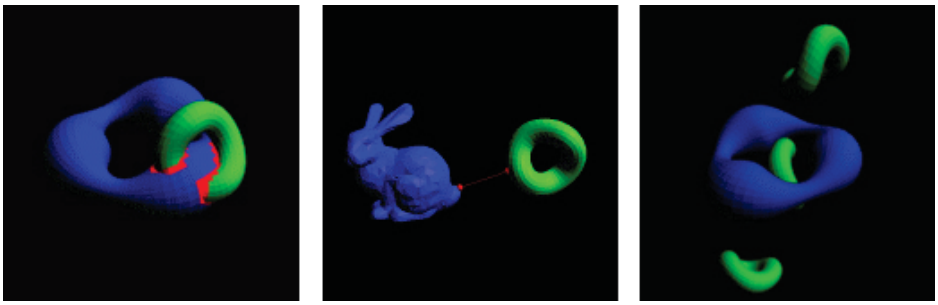

Fig. 4.11 Examples of collision detection between two meshes.

In FCL, objects are represented as instances of the *CollisionObject* class, that encapsulates a *BVHModel* structure, describing the geometry of a mesh, and a *Transform3f* matrix, that represents the pose of the object. On a pair of *CollisionObject* a distance or collision query can be performed.

Furthermore, FCL supports *broadphase* queries between groups of objects that permit to avoid a $O(n^2)$ complexity. To define a group of collision objects it is necessary to create an instance of a *BroadPhaseCollisionManager*. After that, each *CollisionObject* that needs to be inserted in the group must be added to the *BroadPhaseCollisionManager* using its *registerObject()* method. When all the collision objects have been added to their group the *BroadPhaseCollisionManager* must be initialized calling its *setup()*

method. At this point, it is possible to make collision and distance queries between both two collision managers or a collision manager and a single collision object.

It is important to remember that in order to update the poses of the collision objects contained in a collision manager it is possible to directly update the poses for the single objects and then call the *update()* method of the *BroadPhaseCollisionManager*.

The *BroadPhaseCollisionManager* has been exploited in the motion planner developed for the I-SUR project for the collision detection and the distance computation between the robots and the environment. In particular, four collision managers have been defined:

- a manager including the CAD models of the links composing the macro structure of the I-SUR robot and its base structure;
- a manager including the CAD models of the links composing the micro structure of the I-SUR robot and a CAD model of the needle;
- a manager including the CAD models of the links composing the structure of the UR5 robot and the CAD model of the US probe;
- a manager including the CAD models of the objects of the environment.

Algorithm

The component that encapsulate the motion planning function provides to the system a method *planPath* (illustrated in 4.1) that configures the planner and a method *isStateValid* (illustrated in 4.2) that validates a state accordingly with the desired geometric constraints and it is used also by the planner.

Interface

The motion planner component has the following interface:

- **models_folder.** The path to the folder containing the CAD models used by the collision detection;
- **isur_mesh_names.** A list of mesh files associated to the links of the I-SUR robot;
- **UR5_mesh_names.** A list of mesh files associated to the links of the UR5 robot;
- **env_mesh_names.** A list of mesh files associated to objects of the environment, including the model of the phantom;

Algorithm 4.1 pathPlan(q_{start}, q_{goal})

Require: $q_{start}, q_{goal} \in SE(3) \times SE(3)$
1: define two $SE(3)$ state spaces as *ompl::base::SE3StateSpace*
2: construct a compound state space $SE(3) \times SE(3)$
3: set the bounds of the space
4: define a simple setup class as *ompl::geometric::SimpleSetup*
5: set the state validity checking with *setStateValidityChecker*
6: set the state start and goal states with *setStartAndGoalStates*
7: set the planner with *setPlanner*
8: call the *solve* method
9: **if** a valid solution is found **then**
10: store the result in a *MultiAxisPath* structure
11: **else**
12: send an event to signal the failure of the planning
13: **end if**

Algorithm 4.2 isStateValid(q)

Require: $q \in SE(3) \times SE(3)$
1: evaluate the *inverse kinematics* for the I-SUR robot
2: evaluate the *inverse kinematics* for the UR5 robot
3: **if** both the inverse kinematics have solution **then**
4: evaluate the pose of each link of the I-SUR robot with the *forward kinematics*
5: evaluate the pose of each link of the UR5 robot with the *forward kinematics*
6: update the *Transform3f* of each *CollisionObject*
7: update each *BroadPhaseCollisionManager*
8: *distance query* between the I-SUR macro and micro structures
9: *distance query* between the I-SUR macro structure and the UR5 robot
10: *distance query* between the I-SUR micro structure and the UR5 robot
11: *distance query* between the I-SUR micro structure and the environment
12: *distance query* between the UR5 robot and the environment
13: **if** all the distances are inside the threshold values **then**
14: the state is *valid*
15: **else**
16: the state is *not valid* because of collisions
17: **end if**
18: **else**
19: the state is *not valid* because the requested poses are not reachable
20: **end if**

- **macro_link_list.** A list containing the indexes of the links of the macro structure that must be accounted by the collision detection;
- **micro_link_list.** A list containing the indexes of the links of the micro structure that must be accounted by the collision detection;
- **UR5_link_list.** A list containing the indexes of the links of the UR5 robot that must be accounted by the collision detection;
- **min_collision_distance.** The minimum collision distance allowed, used to validate a state;
- **phantom_pose.** The Cartesian pose of the phantom in the scenario;
- **askPlan().** Used to require a motion planning to the component;
- **checkDistancePoses(KDL::Frame ee1_pose, KDL::Frame ee2_pose).** Used to check the minimum collision distance in the scenario given the poses of the end-effectors of the two robots; it encapsulates a call to the private *isStateValid* method;
- **plannedPath.** An output port providing the result of a planning in case of success (*TGL::MultiAxisPath*);
- **isurMeasPose.** The measured Cartesian pose for the I-SUR robot end-effector, used to set the start state;
- **UR5MeasPose.** The measured Cartesian pose for the UR5 robot end-effector, used to set the start state;
- **goalPoses.** The goal poses for the end-effectors of the two robots;
- **events.** An output port on which the component can publish events related to its behavior;

4.5.2 Multi-Arm Cartesian Trajectory Generator

The Cartesian Trajectory Generator is a *Calculation* component and it is used for the generation of a trajectory in the Cartesian space synchronized for both the arms of the setup.

Interface

The component encapsulates functions provided by TGL (see 3.2.3) and its interface is described below:

- **arms_num.**. The number of arms (i.e. Cartesian poses) that must be handled by the trajectory;
- **max_vel.** The maximum velocity allowed for each axis;
- **time_acc.** The acceleration time for each axis;
- **stop_time.** The time in which a stop motion must be executed;
- **sample_time.** The sample time used for the generation of the trajectory;
- **askGenerateTraj().** Used to request to the component the generation of a trajectory: the trajectory is effectively generated only if a valid path is available;
- **askStartMove().** Used to request to the component the start of a motion; after its call, if a valid trajectory is available, the component periodically writes the current Cartesian positions and velocities, accordingly with *sample_time*;
- **askStopMove().** Used to request to the component the stop of the motion; at the moment of its call, if a motion is underway, the component generates a stopping motion from the current state of the trajectory to a state with null velocities accordingly with *stop_time*.
- **plannedPath.** An input port on which a path is received (*TGL::MultiAxisPat*);
- **desCartPose_i.** An output port on which it is written the Cartesian pose (*KDL::Frame)* required by the trajectory at instant t; instances of this ports are created during the configuration of the component accordingly with the value of *arms_num*.
- **desCartTwist_i.** An output port on which it is written the Cartesian twist (*KDL::Twist*) required by the trajectory at instant t; instances of this ports are created during the configuration of the component accordingly with the value of *arms_num*.
- **events.** An output port on which the component can publish events related to its behavior;

As shown in figure 4.12 the component can be in the following states:

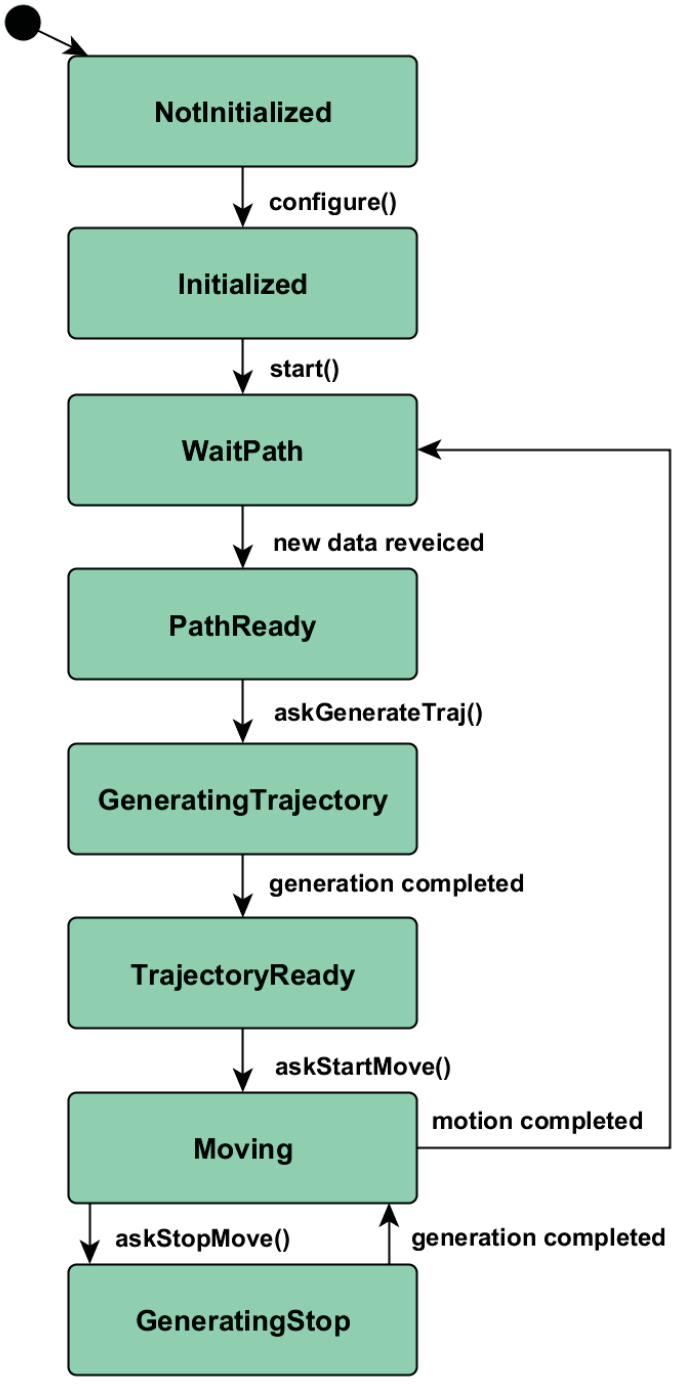

Fig. 4.12 Finite state machine for the trajectory generator component.

- **Not initialized.** The component has been created but not yet configured;
- **Initialized.** The state of the component after its configuration but before its start;
- **Wait path.** In this state the component is running and it is ready to receive a new path;
- **Path ready.** A new path is available;
- **Generating Trajectory.** The generation of a trajectory has been requested and is ongoing;
- **Trajectory Ready.** A valid trajectory is available;
- **Moving.** The component is writing the current trajectory on its ports accordingly with *sample_time*;
- **Generating Stop.** The component is generating a trajectory for the stop of the motion.

4.5.3 Variable Admittance Controller

Admittance control and impedance control [34] are control schemes commonly employed for the implementation of an interaction behavior. Loosely speaking, the choice between this tho control strategies is generally related to the characteristics of the robot on which they will be applied: while the admittance control does not require a dynamic description of the robot and it more suitable for stiff structures, the impedance control requires the knowledge of the dynamic parameters and is more suitable for backdrivable structures.

Given that the robot developed for the I-SUR project has a stiff, not backdrivable structure, it has been decided to implement an admittance control for the architecture. A common interaction model adopted in standard admittance control is the multi-dimensional mass-spring-damper system described by:

$$\Lambda_d \ddot{\tilde{x}} + D_d \dot{\tilde{x}} + K_d \tilde{x} = F_{ext} \tag{4.1}$$

where $x \in^n$ with $n \leq 6$ is the pose of the end-effector obtained from the joint positions $q \in \mathbb{R}^m$, $F_{ext} \in \mathbb{R}^n$ is the external wrench applied to the end-effector, $\tilde{x}(t) = x(t) - x_d(t)$ is the pose error and Λ_d, D_d and K_d are the $n-$dimensional symmetric and positive definite inertia, damping and stiffness matrices characterizing the interactive behavior.

The nature of a surgical task requires a flexible interaction behavior to allow the system to cope with different operative conditions. For example, while penetrating the skin it is

required a stiff behavior but during the insertion the motion must be more compliant, to adapt to the characteristics of the tissues.

Unfortunately, a standard admittance control scheme cannot guarantee a stable behavior in the case of an online variation of its parameters. For this purpose, it has been developed a tank-based time-varying admittance controller that enables to adjust the interactive behavior of the robot while preserving the passivity of the system.

Background on Port-Hamiltonian Systems and Energy Tanks

The variable admittance control has been implemented exploiting the theory behind port-Hamiltonian systems and energy tanks: here it is provided a brief description of this concepts but for further details reference can be made to [35], [36] and [37].

A port-Hamiltonian system provides a framework that allows the description of physical systems. The most common representation of a port-Hamiltonian system is:

$$\begin{cases} \dot{x} = [J(x) - R(x)] \dfrac{\partial H}{\partial x} + g(x)u \\ y = g^T(x) \dfrac{\partial H}{\partial x} \end{cases} \tag{4.2}$$

where $x \in \mathbb{R}^n$ is the state vector and $H(x) : \mathbb{R}^n \to \mathbb{R}$ is the lower bounded Hamiltonian function representing the amount of energy stored in the system. Matrices $J(x) = -J(x)^T$ and $R(x) \geq 0$ represent the internal energetic interconnections and the dissipation of the port-Hamiltonian system, respectively, and $g(x)$ is the input matrix. A port-Hamiltonian system can energetically interact with the external world through a power port that is defined as a pair composed by an input u and an output y (which product is generalized power).

It can be shown (see [35]) that the system either dissipates or stores the power that it receives and, in other terms, that means that a port-Hamiltonian system is passive with respect to the pair (u, y).

The power dissipated by the system can be described by:

$$D(x) = \frac{\partial^T H}{\partial x} R(x) \frac{\partial H}{\partial x} \geq 0 \tag{4.3}$$

and as pointed out in [38] $D(x)$ represents a *passivity margin*. Loosely speaking the *passivity margin* enables the system to absorb the energy generated by non passive actions (e.g. the variation of the parameters describing a visco-elastic coupling) while preserving the passivity.

To exploit this property, it has been introduced the concept of *energy tank*, firstly proposed by [39]. Basically the energy dissipated by the system is stored in a virtual energy tank and

can be extracted from it for implementing a desired control action in a passivity preserving way. The dynamics of a port-Hamiltonian system with energy tank is described by:

$$\begin{cases} \dot{x} = [J(x) - R(x)] \dfrac{\partial H}{\partial x} + g(x)u \\ \dot{x}_t = \dfrac{\sigma}{x_t} D(x) + \dfrac{1}{x_t}(\sigma P_{in} - P_{out}) + u_t \\ y_1 = \begin{pmatrix} y \\ y_t \end{pmatrix} \end{cases} \tag{4.4}$$

where $x_t \in \mathbb{R}$ is the state associated with the energy storing tank and

$$T(x_t) = \frac{1}{2} x_t^2 \tag{4.5}$$

is the amount of energy stored in the tank. $P_{in} \geq 0$ and $P_{out} \geq 0$ are incoming and outgoing power flows that the tank can exchange with other tanks, while the pair (u_t, y_t) represents a power port used by the tank to exchange energy with the external world. The parameter $\sigma \in 0, 1$ is used to bound the amount of energy that can be stored in a tank. The overall variation of energy in the tank can be expressed as:

$$\dot{T} = \sigma D(x) + \sigma P_{in} - P_{out} + u_t^T y_t \tag{4.6}$$

which means that if it is possible to store energy in the tank (i.e. $\sigma = 1$), the contributions of the dissipated power $D(x)$ and of the incoming power P_{in} are stored while the outgoing power P_{out} is released. Moreover energy can be injected or extracted via the power port (u_t, y_t).

The presence of σ can be explained by the necessity of keeping the energy of the tank inside an upper bound $\bar{T}$ to avoid the situation described in [40]. In particular σ is defined as follows:

$$\sigma = \begin{cases} 1 & if\ T(x_t) \leq \bar{T} \\ 0 & otherwise \end{cases} \tag{4.7}$$

where the value of $\bar{T} > 0$ must be tuned depending on the application.

Finally, to avoid singularities in 4.4, it must be $x_t \neq 0$ and for this purpose it is possible to set an arbitrary small threshold $\varepsilon > 0$ that represent the minimum amount of stored energy allowed in the tank.

Algorithm

For the implementation of the variable admittance control the proposed interaction model is the following:

$$\Lambda_d(t)\ddot{\tilde{x}} + D_d(t)\dot{\tilde{x}} + K_d(t)\tilde{x} = F_{ext} \tag{4.8}$$

where $\Lambda_d(t)$, $D_d(t)$ and $K_d(t)$ be the time-varying inertia, damping and stiffness matrices assumed to be symmetric and positive-definite for all $t \geq 0$. See figure 4.13 for an example of control scheme employing the admittance control. However, introducing a variable

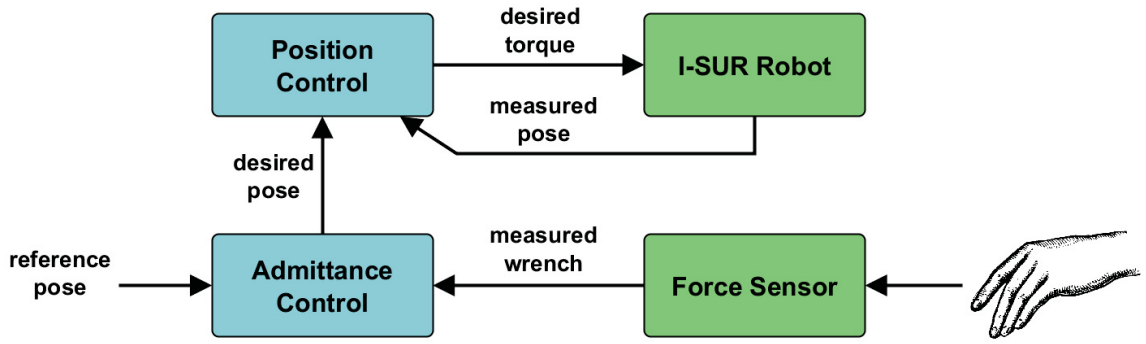

Fig. 4.13 Example of control scheme with admittance control.

interaction model in an admittance control scheme invalidates the passivity of the system as described in [41].

The idea is to reformulate the interaction model 4.8 as a port-Hamiltonian system. First of all $\Lambda_d(t)$ and $K_d(t)$ are defined as sum of a constant and a variable contribution as follows:

$$\begin{cases} \Lambda_d(t) = \Lambda_c + \Lambda_v(t) \\ K_d(t) = K_c + K_v(t) \end{cases} \tag{4.9}$$

The values of the constant parts are arbitrary but once set they will represent the minimum implementable inertia and stiffness.

Considering 4.9 it is possible to describe the interaction model 4.8 as a port-Hamiltonian system augmented with a tank as it follows:

$$\begin{cases} \begin{pmatrix} \dot{\tilde{x}} \\ \dot{\tilde{p}} \end{pmatrix} = \begin{pmatrix} 0 & I \\ -I & -K_d(t) \end{pmatrix} \begin{pmatrix} \frac{\partial H_c}{\partial \tilde{x}} \\ \frac{\partial H_c}{\partial \tilde{p}} \end{pmatrix} + \begin{pmatrix} 0 \\ I \end{pmatrix} F_{ext} + \begin{pmatrix} 0 \\ I \end{pmatrix} w \\ \\ \dot{x}_t = \dfrac{\sigma}{x_t} \tilde{p}^T \Lambda_c^{-1} K_d(t) \Lambda_c^{-1} \tilde{p} - \dfrac{w^T}{x_t} \dot{\tilde{x}} \\ \\ y = \dot{\tilde{x}} \end{cases} \tag{4.10}$$

where $x_t \in \mathbb{R}$ and $T(x_t) = \frac{1}{2} x_t^2$ are the state and the energy function of the tank respectively. The initial state of the tank is set to $x_t(0)$ such that $T(x_t(0)) > \varepsilon$ and the desired input is described by:

$$w(t) = \begin{cases} (-K_v(t)\tilde{x} - \Lambda_v(t)\ddot{\tilde{x}}) & \text{if } T(x_t) > \varepsilon \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

The interaction required by the input w could not preserve the passivity due to the presence of variable inertia and stiffness parameters. Anyway, if there is energy available in the tank it is possible to exploit it for the implementation of the interaction model, otherwise the variable part of the admittance parameters is not considered.

In such a way, the priority is given to the preservation of the passivity of the system. A proof of the passivity of the augmented port-Hamiltonian interaction model with respect to the pair $(F_{ext}, \dot{\tilde{x}})$ can be found here [41].

Interface

The variable admittance controller is a *Calculation* component that encapsulates a variable admittance control and its interface is described as follows:

- **inertia.** A property describing the inertia matrix;
- **stiffness.** A property describing the stiffness matrix;
- **damping.** A property describing the damping matrix;
- **refPose.** An input port providing the reference pose for the controller;
- **refTwist.** An input port providing the reference twist for the controller;
- **measWrench.** An input port providing the external wrench applied to the end-effector;

- **admPose.** An output port on which is published the pose evaluated by the control;
- **admTwist.** An output port on which is published the twist evaluated by the control;

The properties *intertia*, *stiffness* and *damping* are provided in the interface to make them accessible by the architecture and to allow the online reconfiguration of the interaction model.

4.5.4 Passivity and Transparency Layers

As stated before, the I-SUR project aimed to the development of an autonomous surgical robot with the possibility for the surgeon to take over if needed. This requires the implementation of a teleoperation model and the definition of a feasible strategy for the switch to the teleoperated mode. In fact, from both a surgical and a control point of view, during the switch between an autonomous control to a teleoperated mode some problems may arise:

- **Alignment.** An alignment phase enables to cancel the misalignment between the master and the slave and allows an intuitive teleoperation. However, if at the switching time t_s the master and the slave are not aligned, they are both affected by a force produced by their coupling that tries to minimize the misalignment error. The effect is an unexpected bump on both the master and the slave sides: that meas that the surgeon needs to cope with this feedback and, even worse, the slave performs an unconstrained motion. This can be an extremely dangerous scenario in a surgical application in which, most of the time, the robot is interacting with the body of the patient.
- **Constant Offset.** In order to avoid the problems deriving from a misalignment at switching time, a strategy can be imposing the misalignment between the master and the slave as a constant offset between their poses defined as $L = x_m(t_s) - x_s(t_s)$, where $x_m(t_s)$ and $x_s(t_s)$ are respectively the master and slave poses at switching time t_s. In this way, the initial position error between the two is null and the coupling does not produce abrupt reactions. The downside of such approach is that, once this offset is introduced, it is never compensated by the system and that means that the surgeon needs to mentally compensate it while teleoperating the robot.

To avoid such problems, a valid strategy could be first aligning the pose of the master to the pose of the slave and then enable the teleoperation coupling. Unfortunately, this would introduce two inconveniences: first, this would force the surgeon to wait the time necessary for the alignment before being able to operate; secondly, if the surgeon interacts with the master before the completion of the alignment phase, this could delay the procedure and possibly generate an unstable behavior [39].

Algorithm

The teleoperation architecture proposed here aims to embed the advantages of the strategies just described minimizing their drawbacks. With an approach similar to the one followed for the variable admittance control, the teleoperation is modeled as a port-Hamiltonian system as follows:

$$\begin{cases} \begin{pmatrix} \dot{x}_i \\ \dot{p}_i \end{pmatrix} = \begin{pmatrix} 0 & I \\ -I & -R_i \end{pmatrix} \begin{pmatrix} \frac{\partial H_i}{\partial x_i} \\ \frac{\partial H_i}{\partial p_i} \end{pmatrix} + \begin{pmatrix} 0 \\ I \end{pmatrix} F_{ext,i} + \begin{pmatrix} 0 \\ I \end{pmatrix} F_i \\ \quad y_i = \begin{pmatrix} 0 & I \end{pmatrix} \begin{pmatrix} \frac{\partial H_i}{\partial x_i} \\ \frac{\partial H_i}{\partial p_i} \end{pmatrix} = v_i \qquad i = m,s \end{cases} \tag{4.12}$$

where $x_i \in \mathbb{R}^n$, $p_i \in \mathbb{R}^n$ and $v_i \in \mathbb{R}^n$ represent the pose, the momentum and the velocity. H_i is the kinetic energy of the robot and R_i is a symmetric positive definite matrix representing the damping in the system, possibly augmented using local damping injection [35]. F_i is the generalized force due to the bilateral coupling, while $F_{ext,i}$ represents the force due to the interaction with the external world. For the master and the slave this force is indicated by F_h, the force applied by the human, and by F_e, the force applied by the environment, respectively.

It is considered the case in which the surgeon is physically in proximity of the robot and thus it is assumed a negligible communication delay. A common choice is an indirect force feedback teleoperation with a PD coupling (see e.g. [42, 43]) described by:

$$\begin{cases} F_m = -K(x_m - x_s) - B(\dot{x}_m - \dot{x}_s) \\ F_s = +K(x_m - x_s) + B(\dot{x}_m - \dot{x}_s) \end{cases} \tag{4.13}$$

where $K \in \mathbb{R}^{n\times n} > 0$ and $B \in \mathbb{R}^{n\times n} > 0$ are the proportional and the derivative gains respectively. This is equivalent to interconnect master and slave with a (virtual) spring-damper system (see figure 4.14).

To get rid of the initial abrupt forces it can be introduced an offset L that represent a rest length for the spring as follows:

$$\begin{cases} F_m = -K(x_m - x_s - L) - B(\dot{x}_m - \dot{x}_s) \\ F_s = +K(x_m - x_s - L) + B(\dot{x}_m - \dot{x}_s) \end{cases} \tag{4.14}$$

At switching time $L = x_m(t_s) - x_s(t_s)$ is set: this would introduce a constant offset between master and slave. However, as stated before, such offset is undesirable and for this reason L is replaced with a continuous time function $l(t)$ such that $l(t_s) = L$ and $l(t) = 0$ for $t \geq t_s + t_c$.

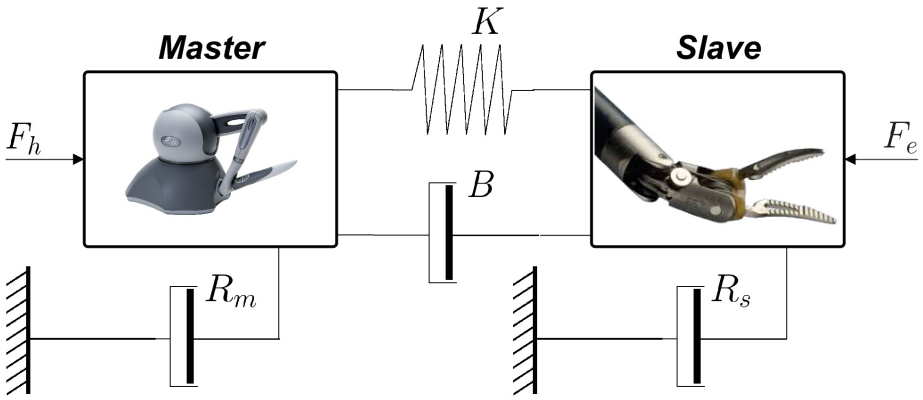

Fig. 4.14 Master and slave interconnection.

The time evolution of $l(t)$ and the amount of time $t_c - t_s$, necessary for completing the compensation, are free parameters that can be set by the designer.

Nevertheless, since (4.14) is a bilateral interconnection, varying the rest length of the spring-like element would generate an elastic force applied to both the master and the slave. While the alignment of the master toward the slave is desired, a movement of the slave toward the master must be avoided because it can be dangerous in the case of direct contact with the patient.

Thus, the master-slave coupling implemented is the following:

$$\begin{cases} F_{md} = -K(x_m - x_s - l(t)) - B(\dot{x}_m - \dot{x}_s) \\ F_{sd} = \alpha(t)[K(x_m - x_s - l(t)) + B(\dot{x}_m - \dot{x}_s)] \end{cases} \tag{4.15}$$

where F_{md} and F_{sd} indicate the desired values of F_m and F_s respectively. The smooth function $\alpha(t) : \mathbb{R} \mapsto [0,1]$ is used for weighting the desired force to be applied to the slave side and it is defined by:

$$\alpha(t) = \begin{cases} \alpha_1(t)\alpha_2(t) & \text{if } t_s < t < t_M \\ 1 & \text{if } t \geq t_M \end{cases} \tag{4.16}$$

where t_M is the first instant of time at which $\alpha_1(t)\alpha_2(t) = 1$. Let $e(t) = \|x_m(t) - x_s(t)\|$ be the norm of the position error and let $\lambda(t) = \|l(t)\|$. The map $\alpha_1 = \alpha_1(e(t)) : \mathbb{R}^+ \mapsto [0,1]$ is a smooth real function defined as:

$$\alpha_1(e(t)) = \begin{cases} 1 & \text{if } e(t) \leq \bar{e}_1 \\ f_1(e(t)) & \text{if } \bar{e}_1 < e(t) < \bar{e}_2 \\ 0 & \text{if } e(t) \geq \bar{e}_2 \end{cases} \tag{4.17}$$

where $f_1(e(t))$ is a non increasing function and $\bar{e}_1 < \bar{e}_2$ are thresholds that can be set by the designer. Similarly, $\alpha_2 = \alpha_2(\lambda(t)) : \mathbb{R}^+ \mapsto [0,1]$ is a smooth real function defined as:

$$\alpha_2(\lambda(t)) = \begin{cases} 1 & \text{if } \lambda(t) \leq \bar{\lambda}_1 \\ f_2(\lambda(t)) & \text{if } \bar{\lambda}_1 < \lambda(t) < \bar{\lambda}_2 \\ 0 & \text{if } \lambda(t) \geq \bar{\lambda}_2 \end{cases} \tag{4.18}$$

where $f_2(\lambda(t))$ is a non increasing function and $\bar{\lambda}_1 < \bar{\lambda}_2$ are thresholds that can be set by the designer. α_1 and α_2 are used to modulate the influence of both the position error and the varying rest length of the spring on the elastic forces affecting the slave. In this way the slave is completely affected by the coupling (i.e. $\alpha(t) = 1$) only when both $e(t)$ and $\lambda(t)$ are small enough and in this case (4.15) becomes equivalent to (4.13).

Unfortunately changing the rest length of a (virtual) spring is not a passivity preserving operation (see [35]). Furthermore, the coupling proposed in (4.15) is asymmetric and this can destroy the passivity of the controller leading to a potentially unstable behavior.

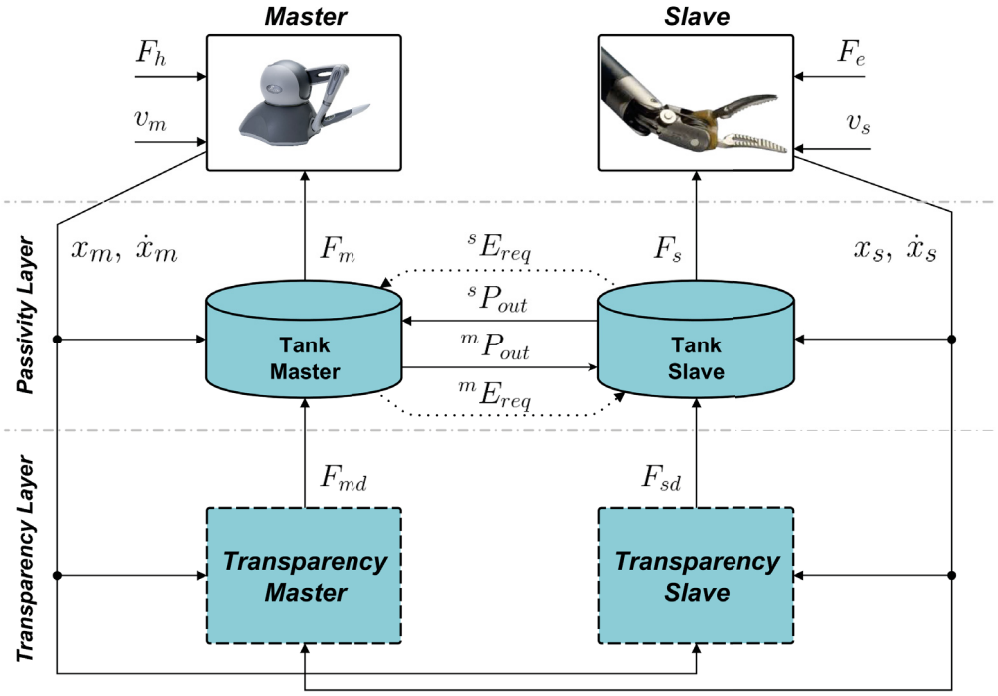

Fig. 4.15 Two-layer architecture for the teleoperation.

To preserve the passivity of the teleoperation system and to preserve the performance of (4.15), it has been exploited the *two-layer* framework proposed in [37]. The architecture can be decomposed into two layers called respectively *Transparency* and *Passivity* layers (see figure 4.15). For this purpose the port-Hamiltonian system previously described is

augmented with a tank than can store energy and then provide it for the implementation of control actions. The augmented system can be described as:

$$\begin{cases} \begin{pmatrix} \dot{x}_i \\ \dot{p}_i \end{pmatrix} = \begin{pmatrix} 0 & I \\ -I & -R_i \end{pmatrix} \begin{pmatrix} \frac{\partial H_i}{\partial x_i} \\ \frac{\partial H_i}{\partial p_i} \end{pmatrix} + \begin{pmatrix} 0 \\ I \end{pmatrix} F_{ext,i} + \begin{pmatrix} 0 \\ I \end{pmatrix} F_i \\ \dot{x}_{t_i} = \dfrac{\sigma_i}{x_{t_i}} D_i(x_i) + \dfrac{1}{x_{t_i}} (\sigma_i {}^i P_{in} - {}^i P_{out}) + u_{t_i} \\ y_i = \begin{pmatrix} v_i \\ y_{t_i} \end{pmatrix} \qquad i = m, s \end{cases} \tag{4.19}$$

where x_{t_i}, $y_{t_i} = x_{t_i}$ and $T_i = \frac{1}{2} x_{t_i}^2$ are the state of the tank, the output associated to the tank and the energy stored in the tank respectively. D_i is the energy dissipated by the robot (that can be augmented by introducing a local damping injection [36]) and ${}^i P_{in}$ and ${}^i P_{out}$ are the power flows that can be exchanged with the other tank.

The desired coupling forces are implemented using the energy stored in the tanks by interconnecting the power port of the tank (u_{t_i}, y_{t_i}) with the power port of the robot (F_i, v_i) using the following power preserving interconnection:

$$\begin{cases} F_i = \dfrac{F_{id}}{x_{t_i}} y_{t_i} = \dfrac{F_{id}}{x_{t_i}} x_{t_i} = F_{id} \\ u_{t_i} = -\dfrac{F_{id}^T}{x_{t_i}} v_i \end{cases} \qquad i = m, s \tag{4.20}$$

The *Transparency* layer calculates the desired coupling forces for master and slave, namely F_{md} and F_{sd}. These forces are then sent to the *Passivity* layer that tries to implement them depending on the energy in the tanks. Master and slave tanks are allowed to exchange energy accordingly with a set of rules:

- while the energy in a tank is below a threshold value ε_i, energy cannot be extracted from the tank (i.e. it is not possible to implement the desired control action);
- while the energy in a tank is below a threshold value ${}^i T_{req}$, the tank can send an energy request ${}^i E_{req}$ the other tanks;
- while the energy in a tank is above a threshold value ${}^i T_{ava}$ a tank is allowed to provide energy to the tanks requesting it;
- while the energy in a tank is above a threshold value ${}^i \bar{T}$ the tank is not allowed to store other energy.

The relation between these thresholds must necessarily be $\varepsilon_i < {}^iT_{req} < {}^iT_{ava} < \bar{T}_i$ (see figure 4.16).

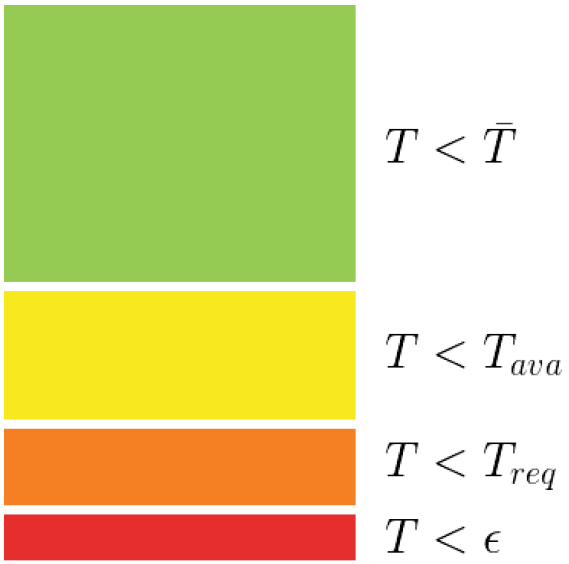

Fig. 4.16 Tank energy levels.

The overall exchange of energy between the tanks in the system can be described as:

$$\begin{cases} {}^mP_{out} = (1-\sigma_m)D_m + {}^sE_{req}\beta_m\bar{P} = {}^sP_{in} \\ {}^sP_{out} = (1-\sigma_s)D_s + {}^mE_{req}\beta_s\bar{P} = {}^mP_{in} \end{cases} \tag{4.21}$$

where ${}^iE_{req}$ is defined as:

$${}^iE_{req} = \begin{cases} 1 & if\ T_i(x_{t_i}) < {}^iT_{req} \\ 0 & otherwise \end{cases} \qquad i = m,s \tag{4.22}$$

and β_i is given by:

$$\beta_i = \begin{cases} 1 & if\ T_i(x_{t_i}) \geq {}^iT_{ava} \\ 0 & otherwise \end{cases} \qquad i = m,s \tag{4.23}$$

$\bar{P} > 0$ is the rate of energy flowing from one tank to the other and it is a design parameter. The bigger is $\bar{P}$, the faster is the energy transfer.

Interface

Both the *transparency* layer and the *passivity* layers are implemented as *Calculation* components. Their interface is quite simple and it is basically corresponding to what shown

in figure 4.15. The transparency layer implements a configurable mass-spring-damper; the passivity layer component is provided with properties that allow to configure the values of ε_i, ${}^iT_{req}$, ${}^iT_{ava}$, $\bar{T}_i$ and of $x_t(0)$.

4.5.5 Puncturing Frame Generator

The puncturing frame generator component is a *Bridge* component. It is used to load preoperative information (e.g., the position of the tumor in the abdomen et al.), the results of the cryoablation planner and the results of the US probe planner. All this data is usually registered in a reference frame related to the abdomen of the patient and needs to be transformed in the task frame.

By design, this component is not reusable in other tasks because it has been implemented to specifically support the puncturing task. However, some flexibility has been provided: the components reads the current state of the task and, with reference to a configurable table, prepares the data that could possibly be required in such state. This enables to easily modifying the behavior of the component for each state of the task. The updated data is then provided to the system on demand.

This is an example of how injecting knowledge about the task inside a component prevents it from being reusable in different contexts. Anyway, this is perfectly acceptable when it happens as a precise choice and not as the consequence of a poor design approach.

Interface

The interface of the component is the following:

- **preoperative_info_file.** The path to the file containing the preoperative information;
- **cryo_plan_file.** The path to the file containing the results of the cryoablation planner (i.e. the poses of each planned needle);
- **US_pose_plan_file.** The path to a file containing the results of the US probe planner (i.e. the poses of the US probe associated to each planned needle);
- **askCurrentGoal().** A method used by the system to ask to this component to publish the available data for the current state of the task;
- **abdomenFrame.** An input port providing the frame of the abdomen in the task frame estimated by the sensing;
- **currentTaskState.** An input port providing the current state of the puncturing task;

- **needleMeasPose.** An input port providing the measured pose of the needle;
- **USMeasPose.** An input port providing the measured pose of the US probe;
- **events.** An output port used to generate events for the reasoning part of the system;
- **desiredPoses.** An output port used to provide the start and goal poses for the current state of the task;

4.5.6 I-SUR Robot Visualizer and Robot Visualizer Bridge

The robot visualizer is a standalone application, developed for debug and simulation purposes, that communicates with the architecture through a dedicated *Bridge* component. The robot visualizer bridge exchanges data with the system through its ports and copies it on a *shared memory* structure to which the robot visualizer has access. Its interface will not be described here because it simply represents a collection of meaningful data selected from the whole architecture.

The I-SUR robot visualizer has been extremely useful during the development of the task: the bridge component through which it communicates with the rest of the system has been implemented to be completely replaceable by the bridge components communicating with the physical robots. Thus, thanks to the visualizer it is possible to test both the logic and the collision avoidance of the architecture without the need of being connected to the actual setup.

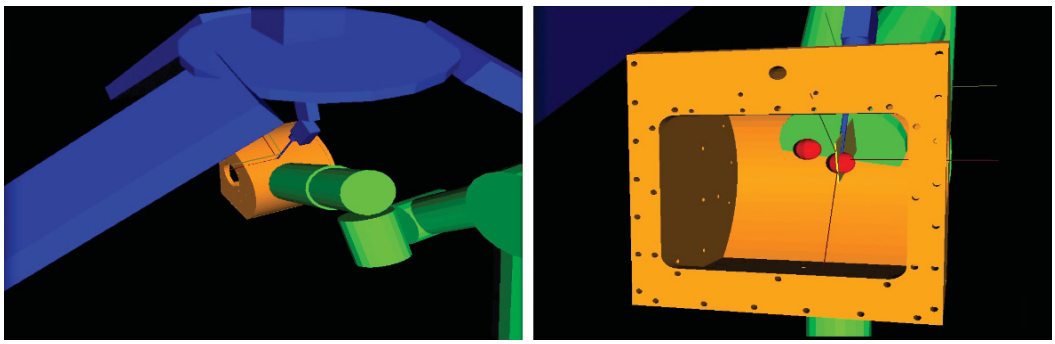

Fig. 4.17 I-SUR robot visualizer for the puncturing task.

The application uses the OpenGL environment (see [44]) to represent a scenario containing the I-SUR robot, the UR5 robot and an abdomen phantom. The meshes used to render the robots are the same employed by the motion planning for the collision detection: in this way there is a perfect match between the planned trajectory and the one visualized in the application. Moreover, the US probe plane is shown as a 3D plane in the visualizer enabling

to verify its alignment with the needle, as requested by the task. An image taken from the visualizer can be observed in figure 4.17.

The visualizer has proven to be an extremely useful tool and it is consider by the author a *best practice*, especially in the cases in which the implementation of the task must proceed in parallel with the development of the hardware (e.g. the robot in the case of the I-SUR project).

4.5.7 I-SUR Robot Bridge

This is a *Bridge* component used to communicate with the low-level control system of the I-SUR robot. It encapsulates the reading and writing functions used to access a *shared memory* on which data is updated. A local, stand-alone server is then in charge of updating the *shared memory* through a *UDP* interface.

Interface

The interface of the component is the following:

- **desCartPose.** An input port used to command a desired Cartesian pose;
- **desJointPosition.** An input port used to command a desired joint position;
- **measCartPose.** An output port on which it is published the last measured Cartesian pose;
- **measJointPosition.** An output port on which it is published the last measured joint position;
- **measWrench.** An output port on which it is published the last measure of the force sensor mounted on the robot;

4.5.8 UR5 Robot Bridge

This is a *Bridge* component used to communicate with the low-level control system of the UR5 robot. It encapsulates the reading and writing functions used to handle the TCP/IP communication with the controller of the robot.

Interface

The interface of the component is the following:

- **desCartPose.** An input port used to command a desired Cartesian pose;
- **desTwist.** An input port used to command a desired twist;
- **desJointPosition.** An input port used to command a desired joint position;
- **measCartPose.** An output port on which it is published the last measured Cartesian pose;
- **measTwist.** An output port on which it is published the last measured twist;
- **measJointPosition.** An output port on which it is published the last measured joint position;
- **measWrench.** An output port on which it is published the last measure of the force sensor mounted on the robot;

4.5.9 OmniPhantom Bridge

This is a *Bridge* component used to communicate with the Omni Phantom haptic device. This component in in charge of exchanging updated data, through a TCP/IP socket, with a stand-alone application running on a Windows platform. The choice of interfacing the device with a Windows platform is due to problems encountered during preliminary tests, performed employing this Firewire device on Linux platforms.

Interface

The interface of this component is defined as follows:

- **host_address.** The IP address of the host;
- **host_port.** The IP port of the host;
- **origin_pose_offset.** Used to set an optional pose offset that is then applied to every measured pose;
- **desWrench.** An input port used to command a desired wrench to the device;
- **measCartPose.** An output port updated with the last measured Cartesian pose;

- **measTwist.** An output port updated with the last measure twist;
- **statusButtons.** An output port describing the last known state of the buttons of the haptic device.

4.5.10 Supervisor

The puncturing supervisor is a *Supervision* component based on a *OCL::LuaComponent*. A *LuaComponent* is essentially a Lua version of a standard OROCOS component and in this case it has been chosen for the possibility of encapsulating a finite state machine implemented with the rFSM Lua module (see subsection 3.1.3). Generally, partial or complete visibility of the architecture is provided to a supervisor and their interaction is possible through the interfaces of the components.

However, manipulating *pure data* directly from a finite state machine it is considered by the author a particularly *bad practice* (see section 2.2) and, for this reason, for the implementation of a task it is suggested to provide the components of the system with *void* operations used to just request services. The data actually considered by the component for the completion of a service is the one available at the moment of its request: if the data exchange presents some criticalities in the system it can be synchronized through a supervisor. In the same way, the completion of a requested service is communicated by the component with an opportune event that is intercepted by the finite state machine.

Interface

The supervisor component has the following interface:

- **fsm_path.** The path of the file containing the description of the finite state machine;
- **inputEvents.** A port on which events are received from the system;
- **outputEvents.** A port on which events are sent to the system;
- **taskState.** A port on which the current state of the task is published.

The implementation of the component allows to create multiple instances of a supervisor, each one potentially running a different finite state machine. For this reason, this component results to be really flexible and it is possible to re-use it for the implementation of supervisors at different levels in the system (e.g., low-level control, hi-level control, task description, configuration, deployment, et al.).

4.6 Deployment

From the point of view of the control of the system, the puncturing task requires three operative modes:

- *autonomous mode with motion planning*;
- *autonomous mode with motion primitives*;
- *teleoperated mode*.

For each of them a specific deployment is required: at each deployment corresponds in fact a defined set of actions available in the system. This set of actions it is not just describable as the sum of the services provided by the single components, but is is instead the result of their composition.

The control architecture for the tree operative modes is here described with reference to the I-SUR robot: the same structure has been replicated for the UR5 robot holding the US probe.

4.6.1 Autonomous Mode with Motion Planning

The set of component deployed is the following:

- **Puncturing Frame Generator.** It provides:
  - *askCurrentGoal()* the component sends data related to the current state of the task to the rest of the system;
- **Motion Planner.** It provides:
  - *askPlan()* the component plans a collision-free path from the current pose to a desired one;
- **Multi-Arm Cartesian Trajectory Generator.** It provides:
  - *askGenerateTraj()* if a path is available, a trajectory is generated accordingly with the motion constraints;
  - *askStartMove()* if a trajectory is available, the component starts to send set points;
  - *askStopMove()* if the component is sending set points, the component generates a stopping motion and starts to send it;

- **Multi-Arm Cartesian Trajectory Generator**;
- **Variable Admittance Control**;
- **I-SUR Robot**;
- **Supervisor**;

The components are connected as shown in figure 4.18.

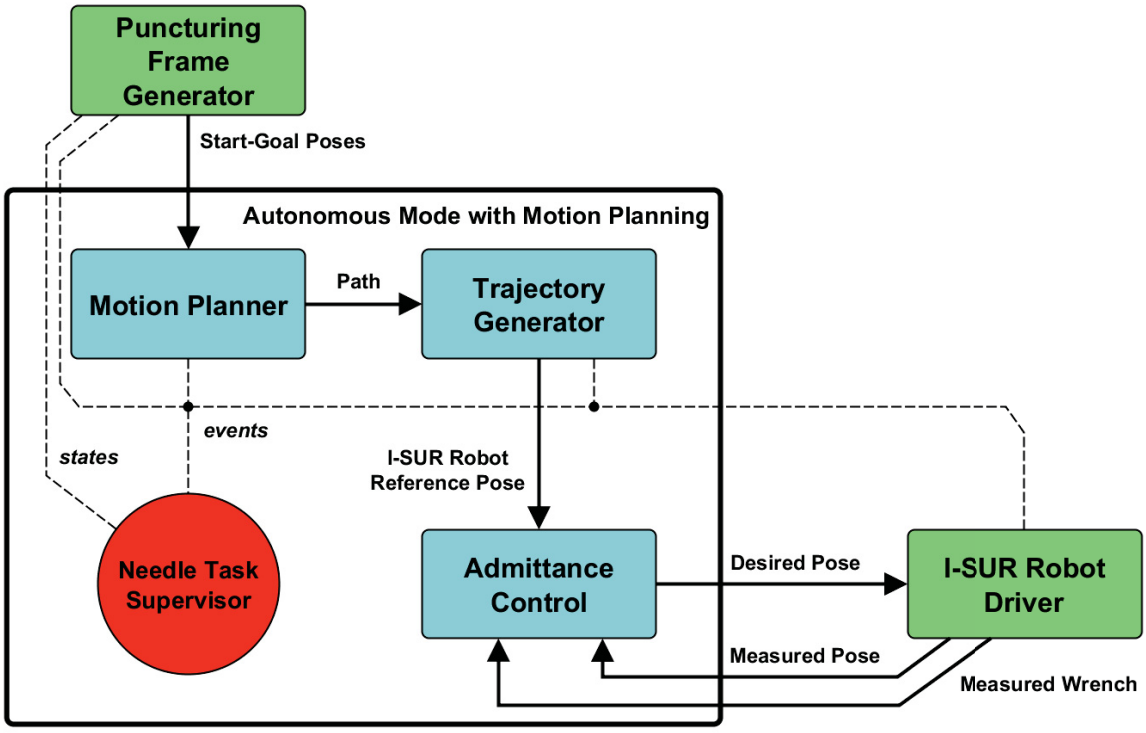

Fig. 4.18 Deployment of the autonomous mode with motion planning for the puncturing task.

4.6.2 Autonomous Mode with Motion Primitives

The deployment for this mode is similar to the other autonomous mode with the only difference that the *Motion Planner* component is removed and the *Puncturing Frame Generator* is directly connected to the *Multi-Arm Cartesian Trajectory Generator* as shown in figure 4.19.

4.6.3 Teleoperated Mode

This operative mode requires a quite different deployment if compared with the previous ones. *Puncturing Frame Generator*, *Motion Planner* and *Multi-Arm Cartesian Trajectory Generator* are not necessary anymore and they are replaced by the following components:

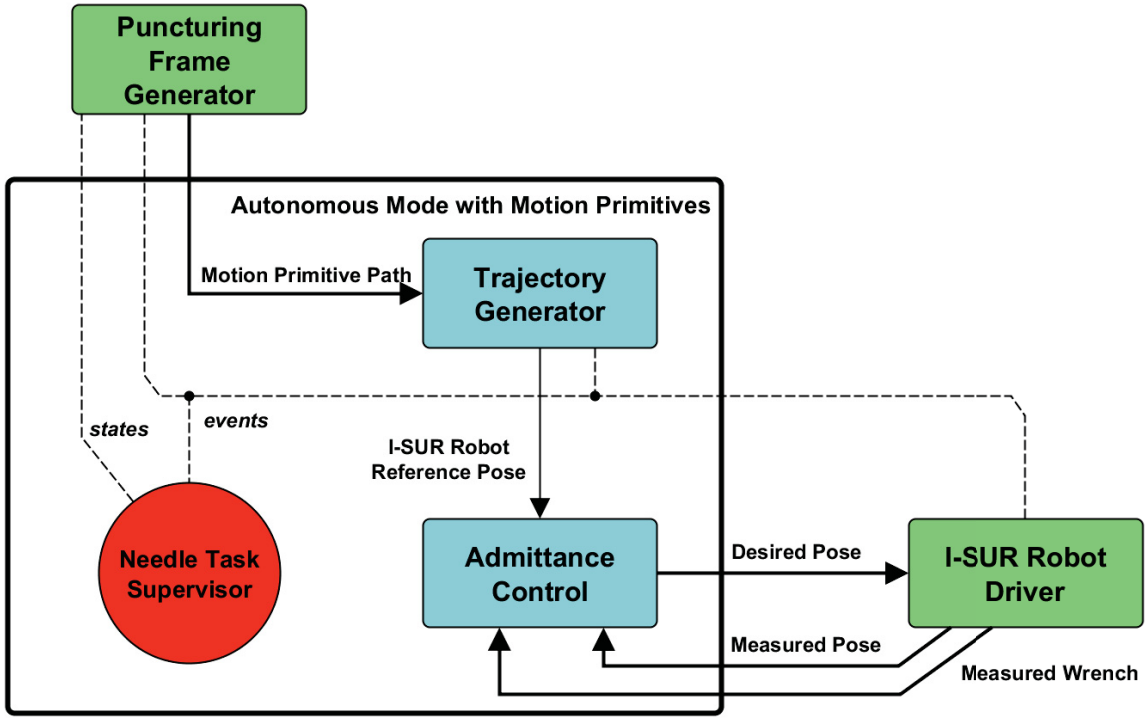

Fig. 4.19 Deployment of the autonomous mode with motion primitives for the puncturing task.

- **OmniPhantom Bridge**;
- **Transparency Layer**;
- **Passivity Layer**;

Figure 4.20 shows a representation of this deployment.

4.7 Configuration

Given a certain deployment, it is possible to modify the behavior of the system modifying its configuration (if the architecture allows it). In the case of the I-SUR project it is necessary to apply different configurations during the insertion of the needle related to the variable admittance control parameters.

The parameters of the variable admittance control typically need to be tuned for a specific task; it is reported here a simple example in which only three configurations are considered:

- **Free-motion behavior.** During a free-motion there are no planned contacts with the environment; however, for safety reasons, the robot is configured to have a compliant

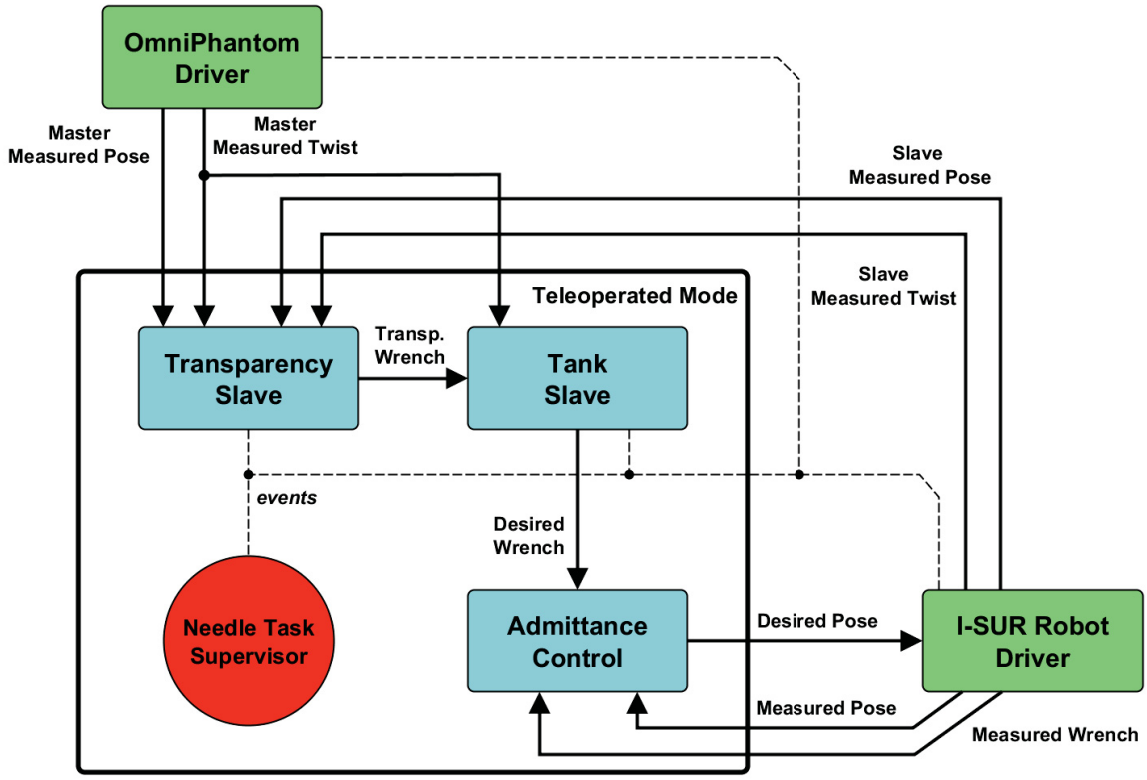

Fig. 4.20 Deployment of the teleoperated mode for the puncturing task (slave side).

behavior in order to reduce the possibility of damages to persons or objects in the case of unexpected collisions;

- **Penetration behavior.** While in contact with the skin the robot must be stiff in order to be able to penetrate it;
- **Needle insertion behavior.** Once the skin has been penetrated, the needle must keep the orientation while being compliant along the insertion axis in order to limit damages in the case in which forbidden regions are touched (e.g., bones, nerves, et al.).

These configurations describe three different kinds of interaction provided by the system and their employment is strictly related to the presence of the variable admittance control of the system. In fact, through the interface of this component it is possible to modify the parameters of the mass-spring-damper model that it encapsulates and then modifying the way in which the robot interact with the environment.

In this case, the name associated to each configuration is representative of a particular behavior required by the task but it could be possible to create a set of *task-independent*, *common-use* configurations and to make them available for the description of a generic task.

4.8 Coordination

Given a description of the task, a set of possible deployments and a set of possible configurations, the system can be driven through a coordination mechanism. As anticipated in 2.2.3, the coordination represents the glue of the whole architecture and the implementation of task, deployment and configuration as three separate entities significantly facilitate this process.

For the puncturing task they have been defined three different control deployments:

- *autonomous mode with motion planning*;
- *autonomous mode with motion primitives*;
- *teleoperated mode*;

and the different configurations:

- *free-motion behavior*;
- *penetration behavior*;
- *needle insertion behavior*.

For every state composing the task it is possible to require an arbitrary combination of deployments and configurations. In figure 4.21 it is shown this mechanism applied to four different phases of the task.

4.9 Results

The proposed architecture has been exploited for the implementation of the task: the two robots have been successfully coordinated during the execution a full puncturing procedure. The main phases of the task can be observed in figure 4.22 and are described as follows:

- **A.** The two robots are in their initial poses for the task; a phantom abdomen has been positioned and registered inside the workspace;
- **B.** After the preoperative planning has been loaded, the UR5 robot starts moving toward the phantom to its planned pose;
- **C.** The UR5 robot has reached the target pose, in contact with the skin; the I-SUR robot is moving to the insertion point;
- **D.** The I-SUR robot has reached the phantom and has started the motion of insertion;

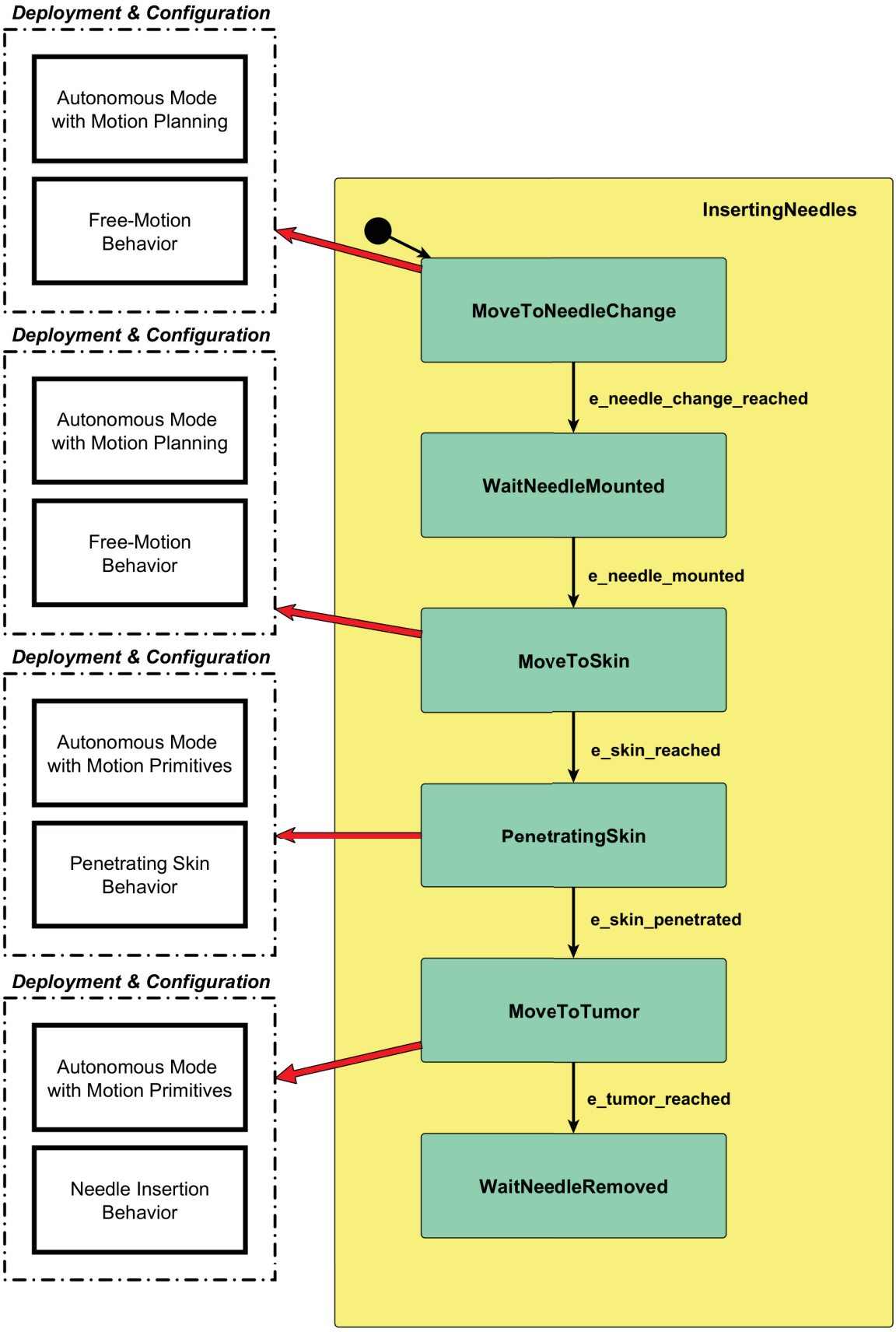

Fig. 4.21 Example of coordination between task, deployment and configuration.

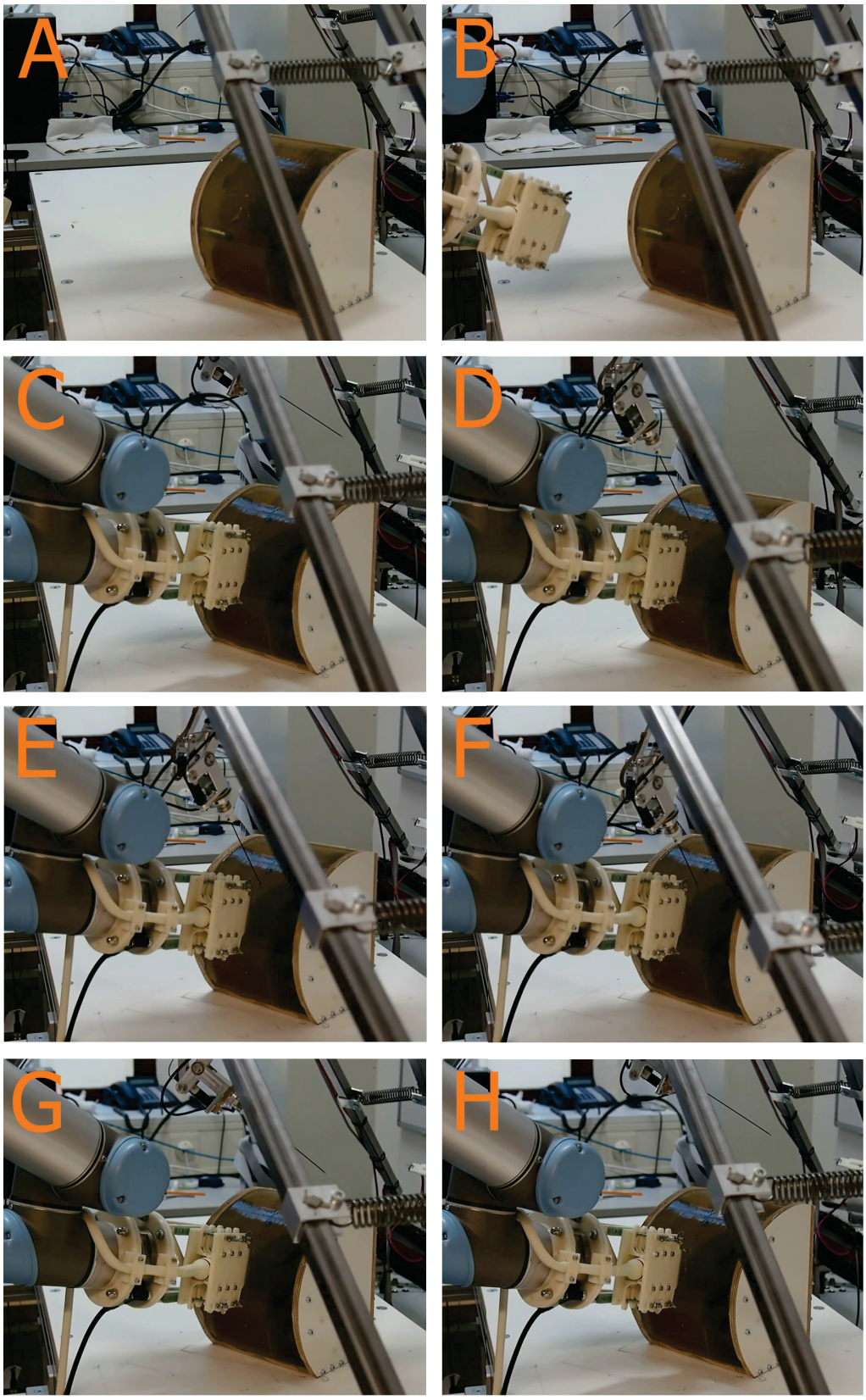

Fig. 4.22 Frames illustrating the execution of the puncturing task.

- **E.** The needle reaches the skin, the configuration of the admittance control is changed accordingly;
- **F.** The skin has been penetrated and the insertion continues; the admittance control is configured to obtain a behavior more compliant;
- **G.** The needle has been extracted and the I-SUR robot performs a collision free motion to the needle change pose;
- **H.** The UR5 moves to the next target pose and the procedure can continue with the insertion of a new needle.

The section of the architecture in charge of the coordination of the system has proven to run smoothly up to a frequency of $1kHz$, without interfering with the real-time control loop. Moreover, other than showing the feasibility of the autonomous surgical procedure, this experiment enabled to verify the flexibility and the reconfigurability of the architecture.

Chapter 5

Case Study: I-SUR *Suturing* Task

In this chapter it will be introduced the suturing task and it will be described in details the architecture developed for its implementation.

5.1 Suturing Surgical Action

Suturing is the act of closing a wound in a biological tissue by means of a thread: different suturing techniques exist, depending on the functional and aesthetic purposes. Many varieties of suture material and needles are available; the choice of sutures and needles, as well as the suturing technique, is determined by location of the lesion, thickness of the tissue, tension exerted on the wound, tensile strength, knot strength, handling and tissue reactivity. Regardless of the specific suture and needle chosen, the basic techniques of needle holding, needle driving and knot placement remain the same. The basic action is to correctly perform a defined number of stitches, depending on the length of the wound; the iterative process, considering a right-forehand handling, is constituted by the following steps:

1. plan the stitching point;
2. insert the needle on the right edge of the wound;
3. pull the thread;
4. insert the needle on the left edge of the wound;
5. pull the thread and return to step 1;

During the I-SUR project it has been considered the case of a planar suture, that is the case of a cutaneous wound described as a linear cut on a planar surface. The surface is

constituted by a single layer made by uniform tissue and the task is to join the two edges of the cut (see figure 5.1).

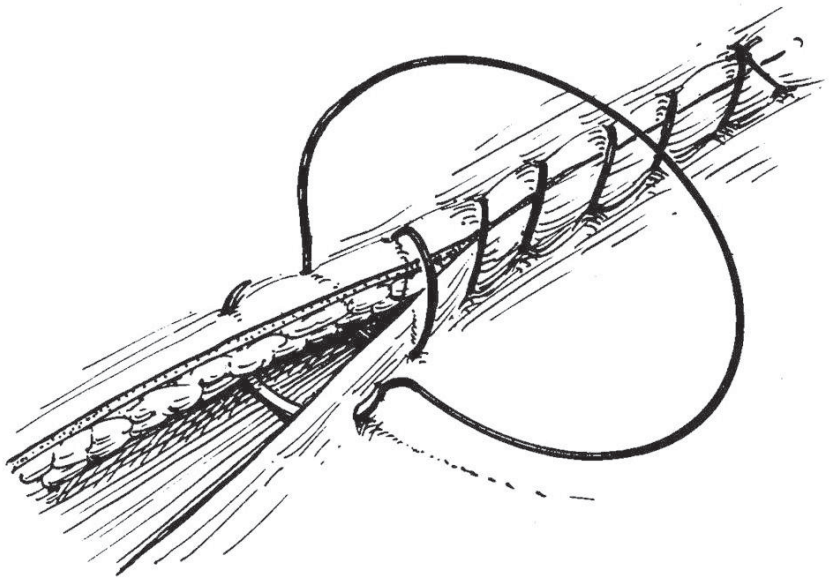

Fig. 5.1 Example of planar suture.

5.2 Task Description

The main goal to be achieved is performing a suturing procedure and it is decomposed into two sub-goals corresponding to the main phases of the task.

5.2.1 Planning

The planning phase requires to analyze the wound and, depending on its geometry, to decide where the next stitch must be applied. Given that the boarder of the wound will change every time that a full stitch is completed and that the two edges of the cut are joined, this procedure needs to be iterated after the application of each stitch.

5.2.2 Applying a stitch

The application of a stitch consists in passing the needle and the thread through both the edges of a wound and then pulling the thread to join them. The motion required for the

insertion of the needle through one edge is almost specular, with the exception that in one case the needle needs to pass from the outside of the wound to the inside, in the other case the opposite.

5.3 Hardware Setup

5.3.1 I-SUR Robot

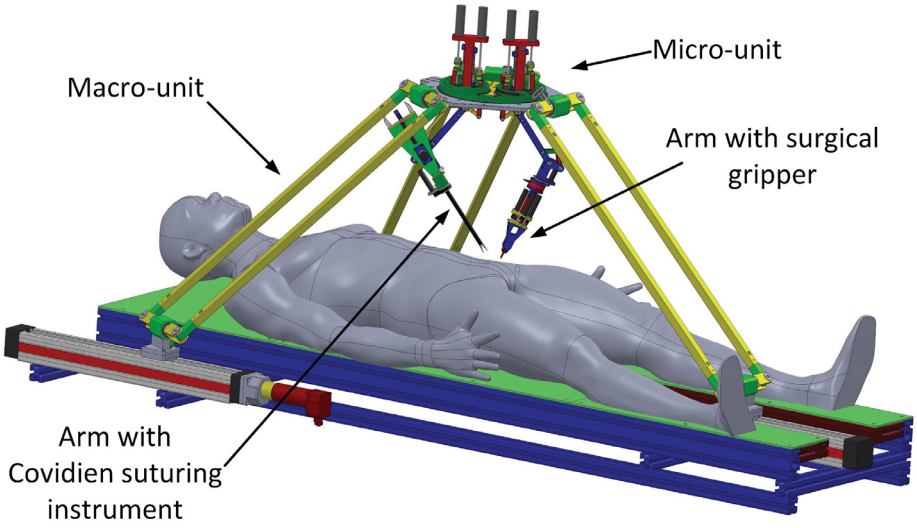

Fig. 5.2 CAD model of the I-SUR robot equipped for the suturing task.

For the implementation of the suturing task the structure of the robot has been updated. The macro structure remains the same while one more arm with 6 degrees of freedom (DoF) has been added to the robot (see figures 5.2 and 5.3).

The updated structure is described as follows:

Macro DoF	4
Right Micro DoF	4
Left Micro DoF	6

The micro arm with 6 DoF is equipped with a gripper while the arm with 4 DoF has been adapted to enable the employment of an Endo Stitch tool developed by Covidien (refer to [45]).

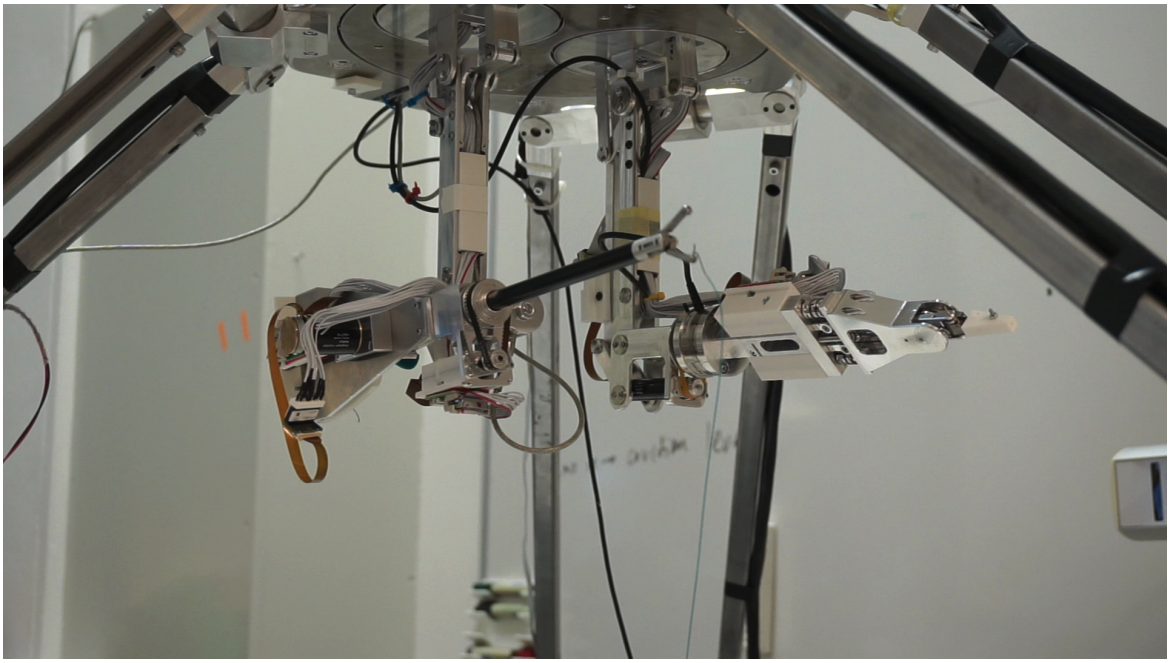

Fig. 5.3 Arms of the I-SUR robot equipped for the suturing task.

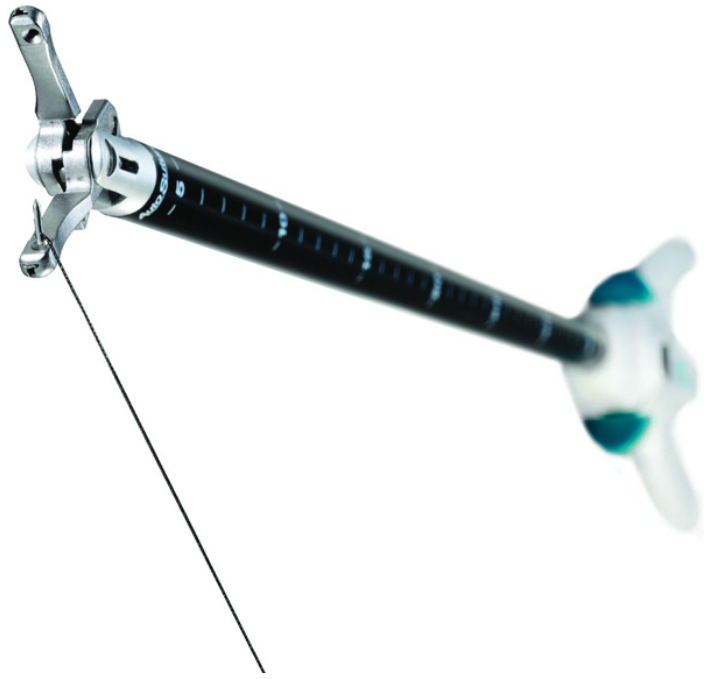

Fig. 5.4 Endo Stitch tool produced by Covidien.

Endo Stitch Suturing Tool

The Endo Stitch tool, produced by Covidien (see figure 5.4), is a suturing instrument used in advanced laparoscopic procedures that require endoscopic suturing and knot tying. It is a one-handed device and it allows to easily transfer the needle within the jaw just acting on a switch. The fact that it can be managed with one arm makes it particularly well suited for an automated procedure and that is the main reason for which it has been chosen.

5.3.2 AtiNano43 Sensor

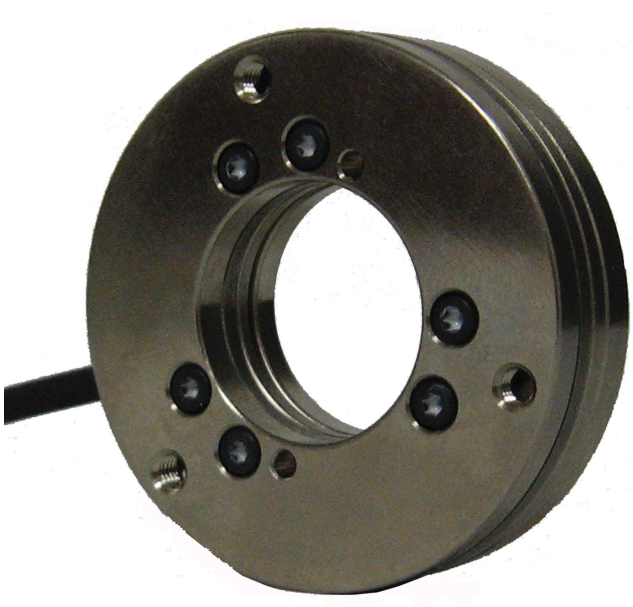

Fig. 5.5 AtiNano43 force and torque sensor.

In order to maintain the force sensing also on the arm holding the Endo Stitch tool, a different kind of sensor has been employed. The AtiNano43 has a center hole that allows the assembling of the tool.

The sensor has the following specifications:

Weight	0.0387 kg
Diameter	43 mm
Height	11.5 mm
Overload Fxy	±300 N
Overload Fz	±380 N
Overload Txy	±3.2 Nm
Overload Tz	±4.6 Nm
Stiffness X-axis & Y-axis forces (Kx, Ky)	$5.2x10^6$ N/m
Stiffness Z-axis force (Kz)	$5.2x10^7$ N/m
Stiffness X-axis & Y-axis torque (Ktx, Kty)	$7.7x10^2$ Nm/rad
Stiffness Z-axis torque (Ktz)	$1.1x10^3$ Nm/rad
Resonant Frequency Fx, Fy, Tz	2800 Hz
Resonant Frequency Fz, Tx, Ty	2300 Hz

5.3.3 Leap Motion

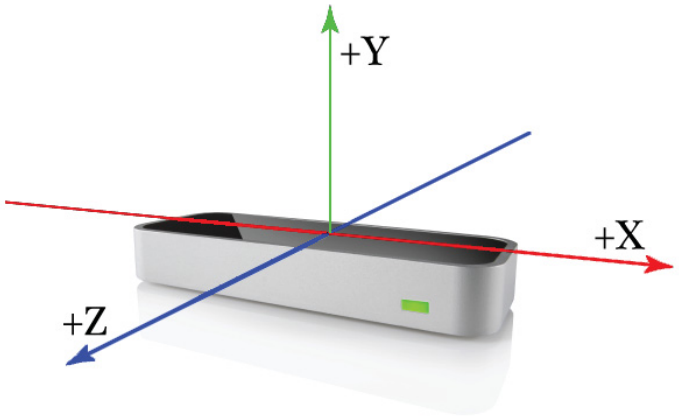

Fig. 5.6 Leap Motion device.

Leap Motion [46] is a device that enables the tracking of both arms and all the fingers (see figure 5.6). It is equipped with two cameras and three infrared LEDs. The cameras track the infrared light with a wavelength of 850 nanometers, which is outside the visible light spectrum. The images acquired by the camera are elaborated to obtain data about the tracking of the hands, and then these information are streamed via USB (see figure 5.7).

The specifications for the device are summarized as follows:

Fig. 5.7 Leap Motion tracking two hands.

Weight	0.045 kg
Height	13 mm
Width	13 mm
Depth	76 mm
Frame Rate	200 fps
Communication	USB

5.4 Developed Components

The implementation of the surgical required minimal adjustments from the point of view of the code. Indeed, the only two components modified are the *Motion Planner* and the *Robot Visualizer* due to the fact that the UR5 has been replaced by a second micro arm mounted directly on the I-SUR robot.

5.4.1 Motion Planner

The fact that the I-SUR robot has been equipped with a second arm introduces deep changes in its kinematics.

On a side, a motion of the macro structure translates both the micro arms; on the other side, only the micro structure with 6 degrees of freedom is capable of translational motion. That means that if it is required a movement of the arm holding the Endo Stitch tool (the one

with 4 degrees of freedom) it is necessary to move the whole macro structure: doing so even the other arm is moved.

From the point of view of the motion planning algorithm employed nothing changes, with the exception that instead of calling the kinematics of two separate robots it is now called a single solver that provides a solution for both the end-effectors of the I-SUR robot. In algorithm 5.1 it is shown the updated version of the algorithm employed for the validation of the sampled states.

Algorithm 5.1 isStateValid(q)

Require: $q \in SE(3) \times SE(3)$

evaluate the *inverse kinematics* for the I-SUR robot
if the inverse kinematics have solution **then**
    evaluate the pose of each link of the I-SUR robot with the *forward kinematics*
    update the *Transform3f* of each *CollisionObject*
    update each *BroadPhaseCollisionManager*
    distance query between the macro structure and the right micro structure
    distance query between the macro structure and the left micro structure
    distance query between the two micro structures
    distance query between the right micro structure and the environment
    distance query between the left micro structure and the environment
    if all the distances are inside the threshold values **then**
        the state is *valid*
    else
        the state is *not valid* because of collisions
    end if
else
    the state is *not valid* because the requested poses are not reachable
end if

The configuration of the I-SUR robot employed for the suturing task introduces a problem related to the redundancy of the structure. In fact, every time a new sample is considered by the planning algorithm it is validated by means of the inverse kinematics of the robot that is used to both describe the reachability of a sample $q \in SE(3) \times SE(3)$ and the absence of collisions.

Due to the redundancy of the structure, starting from similar samples (i.e. one near to the other in the considered sample space) it is possible the convergence of the solver to two completely different configurations of the robot. This kind of problem clearly does not occur when the planning is executed in the joint space and the samples are validate using just the forward kinematics. An another case in which this does not represent a problem is when it is available the closed form solution for the inverse kinematics of the structure.

In the case of the I-SUR project, to overcome the possible problems introduced by the redundant structure, it has been chosen the approach described as follows:

- the structure is split into to sub-structures: one is obtained considering the macro structure and the micro structure holding the Endo Stitch, the other one is constituted by the remaining micro structure. The reason of this choice is that the micro structure holding the Endo Stitch has only 4 DoF and it can be translated only through the macro structure, while the other micro structure has 6 DoF and then it can be both translated and orientated;
- given that a translation of the macro structure would translate both the micro structures, priority is assigned to the motion of the arm holding the Endo Stitch;
- given a sample $q \in SE(3) \times SE(3)$, it is first evaluated a solution for the inverse kinematics of the structure obtained considering the macro structure and the micro structure holding the Endo Stitch; this sub-structure presents redundancies that have been handled considering that if the orientation of the end-effector of the right micro-unit is expressed in terms of Euler ZYX angles (i.e. *roll*, *pitch* and *yaw*) each attitude angle is directly related to either a single or at least a pair of joint positions. In particular, the pitch angle is given by the sum of q_6 and q_7 joint angles, the yaw is equal to $q_4 + q_5$ and the roll angle is exactly q_8;
- once a solution for the first sub-structure has been found, the relative pose of the end-effector of the macro structure is considered as the base pose of the other micro structure that is then evaluated using a closed form solution.

This approach results in a inverse kinematics solution for the whole robotic structure that is equivalent to a closed form solution and that is then feasible for the path planning algorithm.

5.4.2 State Validator

The *State Validator* is a *Calculation* component developed to support the teleoperation of the I-SUR robot modified for the surgical task. In fact, the motion of the robot needed by the suturing task, while in teleoperated mode, is more constrained than in the case of the puncturing task. This is because a motion of the macro structure induces a motion of both the micro arms and the right arm can be translated only using the macro structure.

In order to guarantee that the motion commanded by the teleoperation coupling leads to a valid state of the system (i.e. a reachable and collision free state), this component requires the *isStateValid* method from the interface of the *Motion Planner* and uses it to validate the desired set-point before actually passing it to the robot.

5.4.3 I-SUR Robot Visualizer

In order to allow tests and debug procedures on the whole system without the need of working with the full physical system and with the prototype of the robot that was still in development, even in the case of the suturing it has been used a visualization tool. As for the *Motion Planner* this application required to be updated with the new kinematics of the robot and with a new set of meshes.

Moreover, in order to provide some feedback during the tests performed in teleoperation mode, the application has been integrated with the *State Validator* component and for each pair of commanded poses it displays the correspondent minimum collision distance in the system and information about the validity of the current set-points (see figures 5.8 and 5.9).

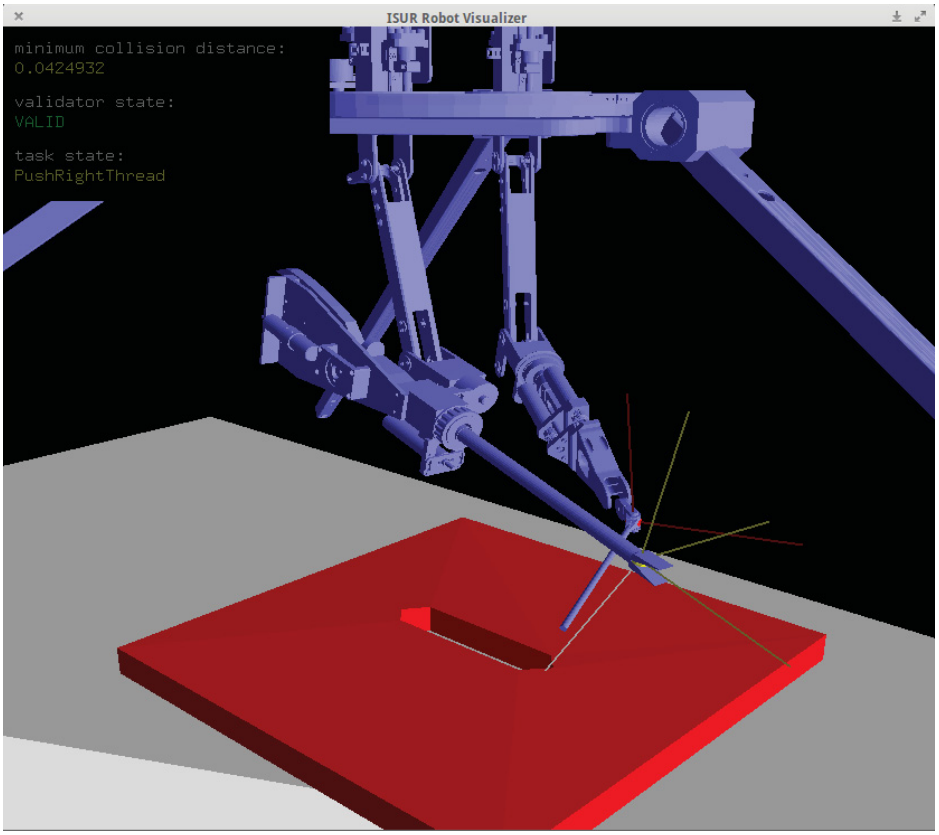

Fig. 5.8 Robot visualizer used in autonomous mode.

5.4.4 Leap Motion Driver

This is a *Driver* component that encapsulates many of the functions provided by the API of the device. Through this component it is possible to read the data coming from a *Leap Motion* device and to make them available to the rest of the system.

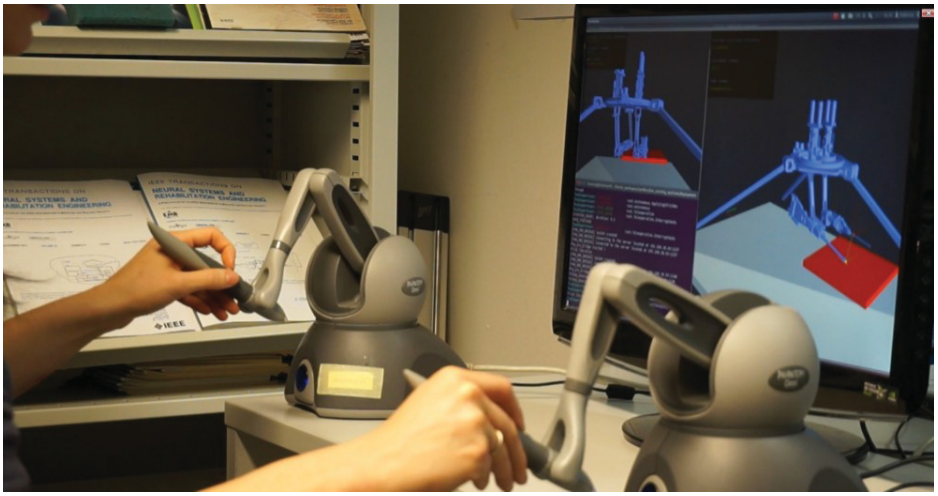

Fig. 5.9 Robot visualizer used in teleoperated mode with two Omni Phantom.

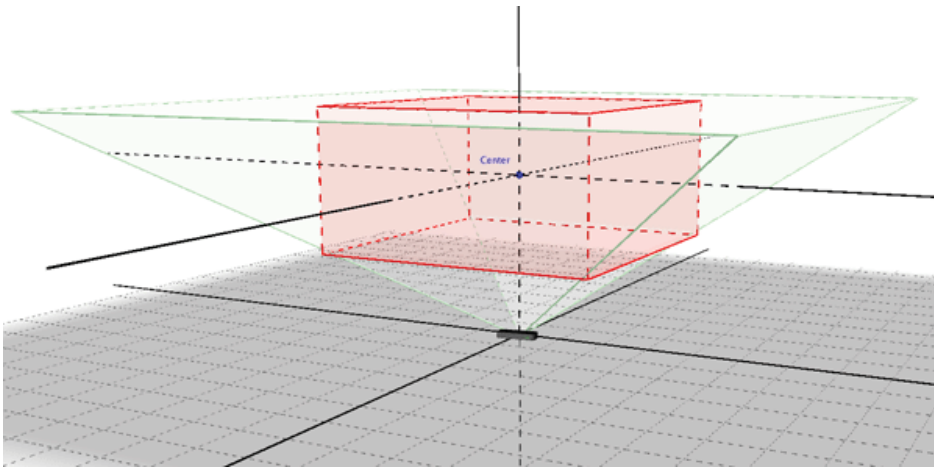

Fig. 5.10 Leap Motion workspace with an example of interaction box.

Interface

The interface of the component is described as follows:

- **interaction_box.** This property allows to define a virtual interaction box contained in the actual workspace of the device; every measured position that would be outside of this box is instead limited to the border of the box (see figure 5.10);
- **workspace_offset.** This property allows to apply a transformation to each measured pose; it is used to change the reference frame of the device;
- **workspace_scale.** This property enables to scale the measured poses by a scalar factor; it is generally used to increase or decrease the sensibility of the device;
- **measRightPose.** The output port on which it is published the last valid right hand pose;
- **measLeftPose.** The output port on which it is published the last valid left hand pose.

Thanks to the flexibility of the architecture it has been possible to use this component in alternation with the Omni Phantom driver without the need of modifying any code, but simply replacing one driver component with the other.

5.5 Task Formalization

As for the puncturing case, the task has been described using rFSM. In this case it has been used a single finite state machine described in figure 5.11.

It follows a description of the main states of the task:

- **WaitStitchesPlan.** In this state the system is waiting for the planning of the stitches poses from the sensing module;
- **AutonomousMode.** It describes the procedure for the autonomous application of one stitch;
- **GetStitchTargetPose.** In this state it is loaded the pose of the current stitch to be applied from the list provided by the stitches plan;
- **ApplyRightStitch.** It describes the procedure for the autonomous application of the first half of the stitch;

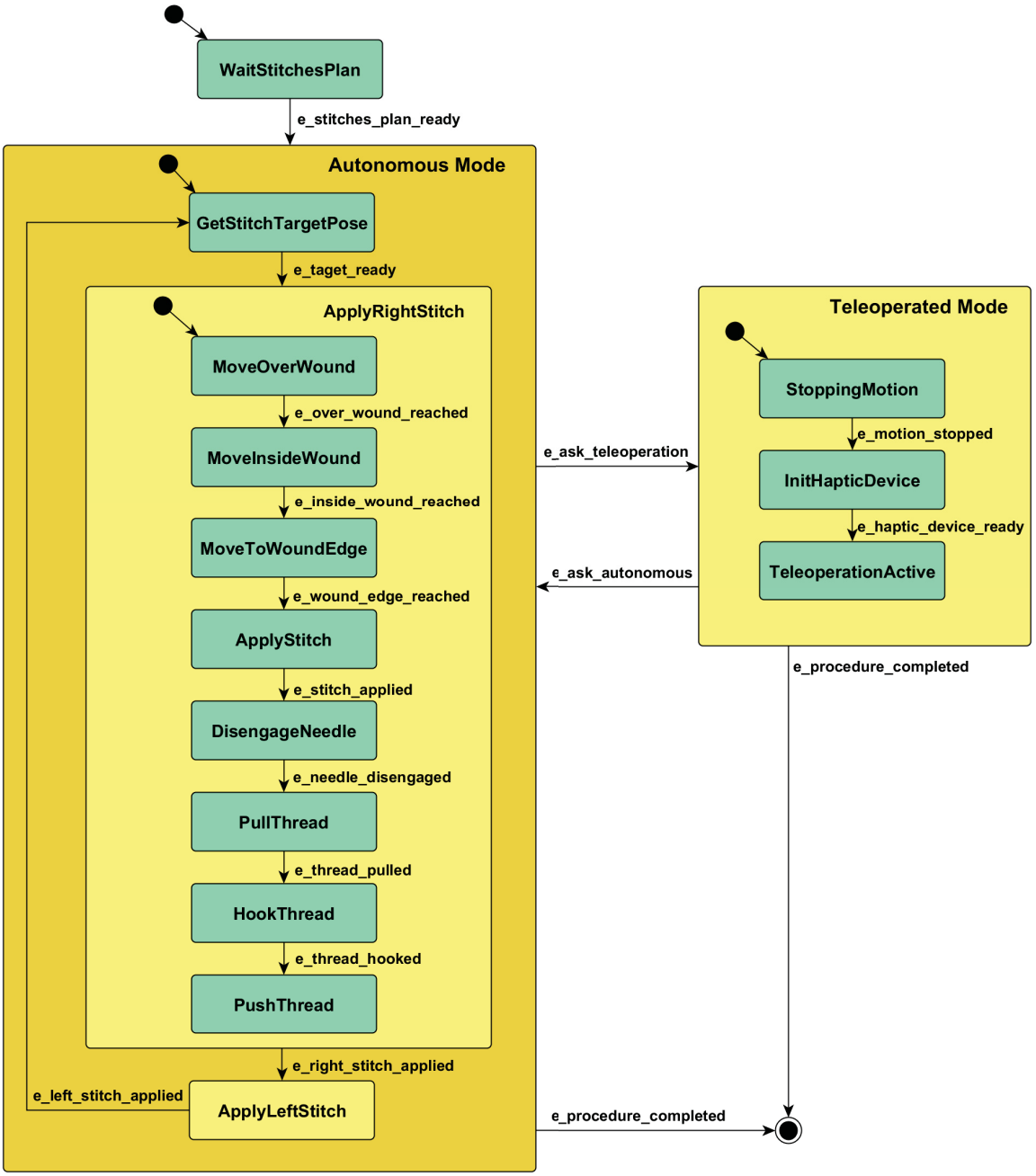

Fig. 5.11 Finite state machine for the suturing task.

- **MoveOverWound.** In this state the needle is moved over the target pose keeping a certain distance from the wound,this is an online planned motion;
- **MoveInsideWound.** It is performed a motion primitive and the tip of the Endo Stitch reaches the inside of the wound;
- **MoveToWoundEdge.** It is performed a motion primitive that rotates the tool until the needle reaches the position at which the stitch will be applied;
- **ApplyStitch.** The Endo Stitch is actuated, the stitch is applied on one side of the wound;
- **DisengageNeedle.** It is performed a motion primitive that enable the needle to be disengaged from the skin;
- **PullThread.** The thread is pulled by the arm holding the Endo Stitch;
- **HookThread.** The other arm, equipped with a gripper holding a stick, perform a primitive movement that enables to hook the thread;
- **PushThread.** Once the thread has been hooked, it is pushed away from the wound to ensure that it will not interfere with the application of the next stitch;
- **ApplyLeftStitch.** It describes the procedure for the autonomous application of the second half of the stitch; its internal states are similar to the state *ApplyRightStitch*;
- **TeleoperatedMode.** In this state it is executed the switch to the teleoperated mode;
- **StoppingMotion.** The autonomous motion of the robot is interrupted generating a stopping trajectory;
- **InitHapticDevice.** The haptic devices are initialized and enabled;
- **TeleoperationActive.** In this state the teleoperation if finally active.

5.5.1 Deployment

From the point of view of the deployment, the set of components is basically the same employed for the puncturing with the insertion of the new *State Validator* between the *Variable Admittance Control* and the *I-SUR Robot Driver*. An example of the control scheme used for the implementation of the suturing task is provided in figure 5.12 and it can be confronted with the analogous scheme provided for the puncturing task (see figure 4.18).

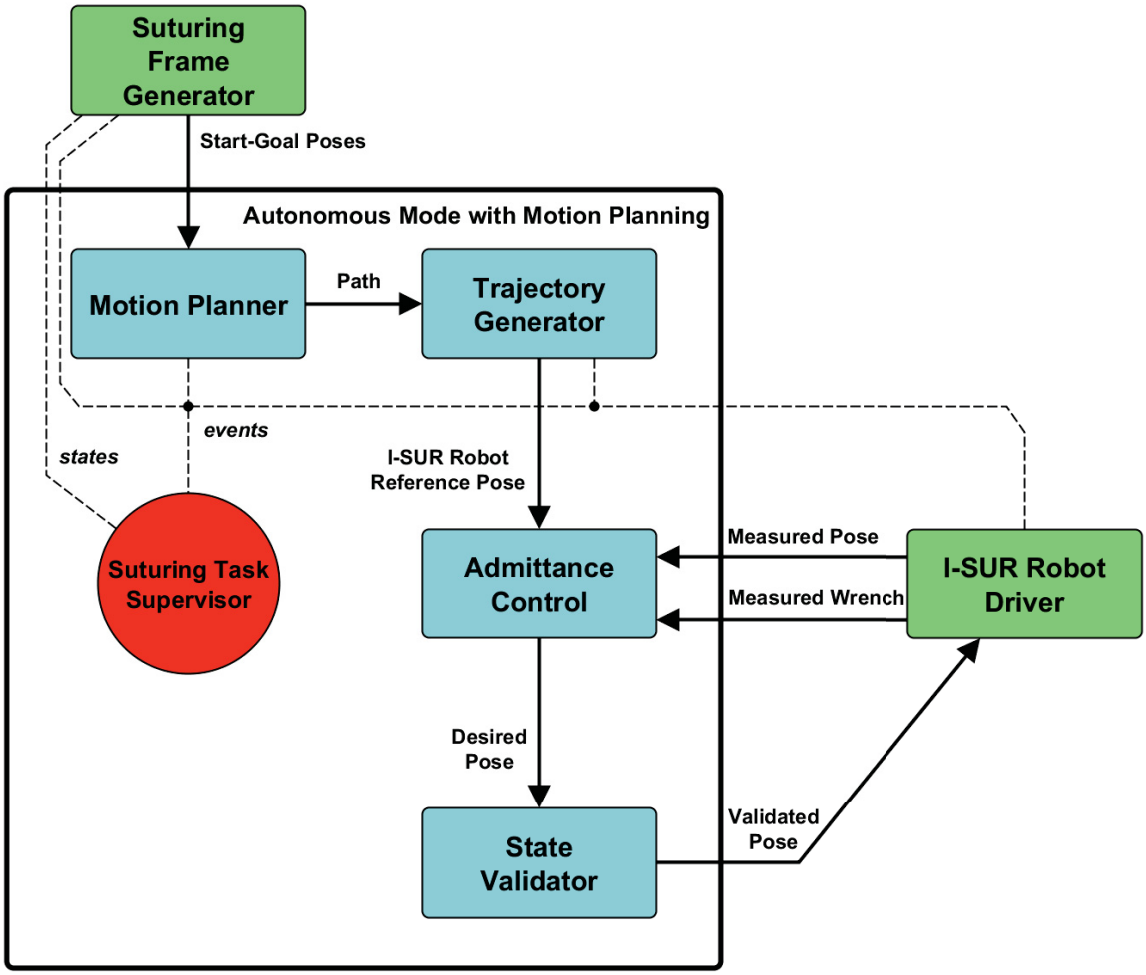

Fig. 5.12 Deployment of the autonomous mode with motion planning for the suturing task.

5.5.2 Configuration

From the point of view of the configuration, the suturing task requires only two kinds of interaction. The implementation of a planar suture requires a less complex model of interaction if compared with the puncturing case. In fact, the phantom used for the experiments is made of only two layers, built using the same material but with different colors. Because of that, the set of required behavior is quite limited and it is described as follows:

- **Free-motion behavior.** Basically the same employed for the puncturing task: the robot is configured to have a compliant behavior in order to reduce the possibility of damages to persons or objects in the case of unexpected collisions;

- **Contact behavior.** This configuration is used to describe a more stiff behavior of the robot and it is basically used only while pulling the thread in order to join the two edges of the wound.

5.6 Results

Due to an incomplete calibration of the robot, it has not been possible to execute a full suturing procedure using the real-system. Anyway, it has been possible to test the whole task through the *I-SUR Robot Visualizer* tool. The main phases of the task are shown in figure 5.13 and are described as follows:

- **A.** The tow arms of the I-SUR robot are in their initial position;
- **B.** After an online planning, the tho arms are moved over the wound;
- **C.** The tip of the Endo Stitch is positioned inside the wound;
- **D.** The Endo Stitch is rotated until it touches one edge of the wound and then the first half of the stitch is applied;
- **E.** Both the arms are moved following motion primitives to pull the thread;
- **F.** The left arm, holding a metal stick, is moved around the thread and positioned behind it;
- **G.** The left arm pushes the thread to ensure that it will not interfere with the application of the next stitch;
- **H.** The two arms are moved back over the wound and the robot is ready to apply the second half of the current stitch;

As for the case of the puncturing task, the architecture has been tested and it proved to enable a smooth deployment and an easy reconfiguration. In order to further demonstrate the efficacy of the design patterns developed during the duration of the I-SUR project, in the next chapter it will be described the implementation of a similar control system on a completely different hardware setup, a retrofitted industrial robot.

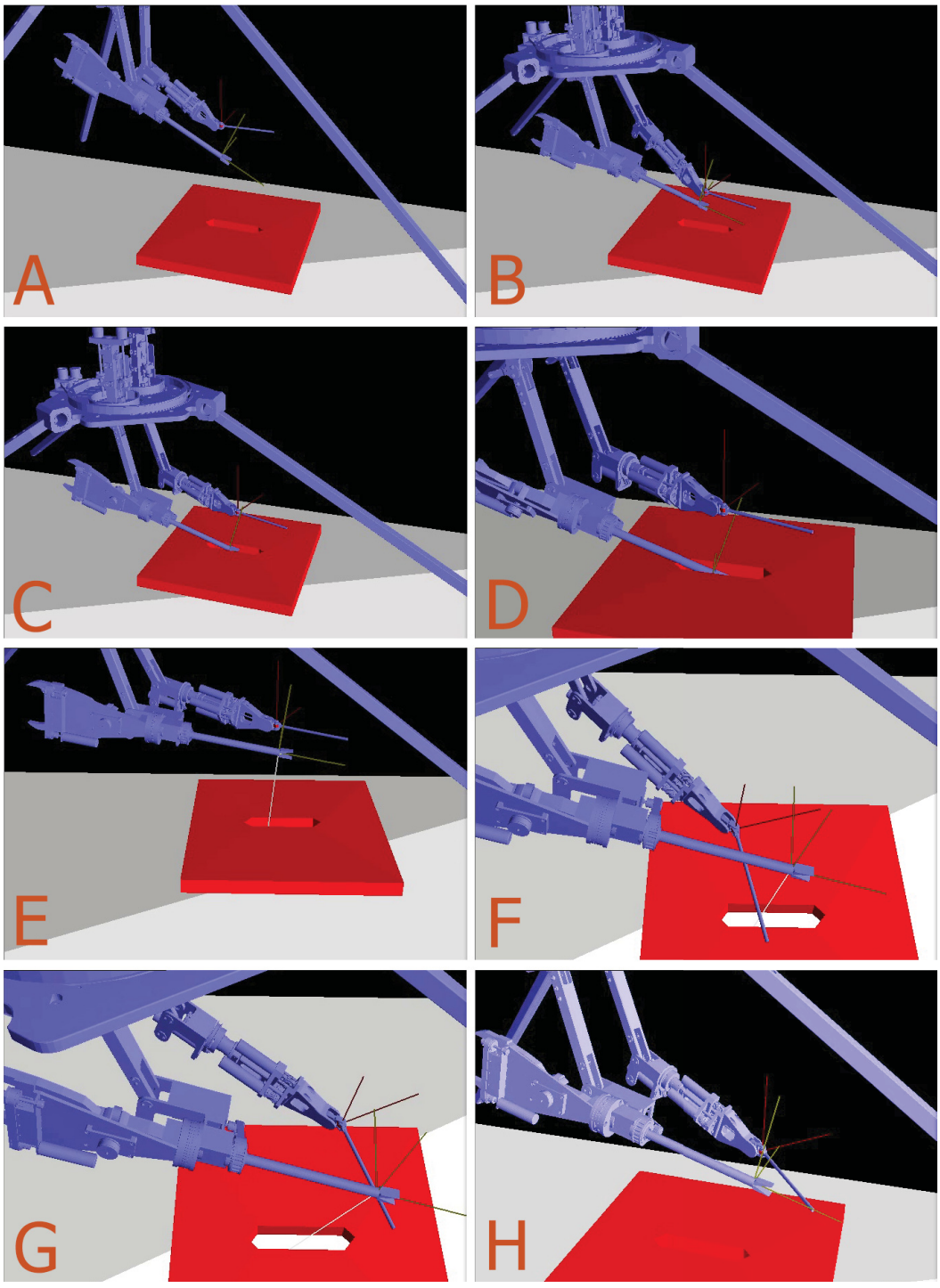

Fig. 5.13 Frames illustrating the execution of the suturing task.

Chapter 6

Case Study: System Architecture Design for a Retrofitted Puma260

This chapter is about the design of a flexible control architecture for a retrofitted Puma260. The system has been developed with the purpose of reproducing the control scheme employed for the implementation of the puncturing task by the I-SUR project on a Puma 260 robot, proving the reusability of most of the components.

6.1 Hardware Setup

It follows a description of the hardware components of the setup.

6.1.1 Retrofitted Puma 260

The Puma 260 is an industrial 6-axis anthropomorphous manipulator firstly produced by Unimation during the 1980s, whose main characteristics can be listed as follows:

Weight	13.2 kg
Degrees of freedom	6
Drive	Electric DC servos
Load capacity	1.0 kg

Along the years they have been released several versions of the controller:

- *Mark I*
- *Mark II*

Fig. 6.1 Puma 260 with UNIVAL controller.

- *Mark III*
- *UNIVAL*

Every controller consists of an interface board, a control board and a power board. The *Mark* series allows an easy retrofit thanks to the possibility to bypass the original control board and to send commands directly to the power board. The robot used in the setup was equipped instead whit an *UNIVAL* controller: in this case the retrofit requires the replacement of the whole controller. For this purpose, it has been chosen to employ a set of *Gold Solo Whistle* EtherCAT drivers, produced by *Elmo* (see [47]).

Gold Solo Whistle Driver

The *Gold Solo Whistle* is a compact digital servo driver produced by Elmo. Its specifications are listed in the following table:

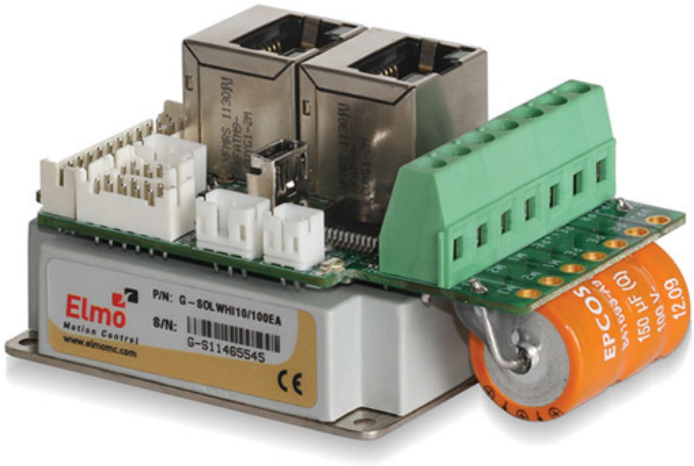

Fig. 6.2 Gold Solo Whistle Drive.

Dimensions	72.4 x 46.5 x 35.8 mm
Weight	106 g
Minimum supply voltage	12 V
Nominal supply voltage	85 V
Maximum supply voltage	95 V
Maximum continuous power output	1.6 kW
Maximum peak power	3.2 kW
Maximum output voltage	> 95% of DC bus voltage at f = 22 kHz
Amplitude sinusoidal/DC continuous current	20 A
Sinusoidal continuous RMS current limit (I_c)	14.1 A
Peak current limit	$2 \times I_c$
Communication	EtherCAT, CANopen

The new controller of the robot is equipped with six Elmo drivers that allow position, velocity and torque control. The drivers can be commanded by an EtherCAT master that in this case is implemented on a computer using the *Simple Open EtherCAT Master* library (SOEM, see [48]).

6.1.2 FTSens Sensor

The FTSens is a force/torque sensor produced by the Italian Institute of Technology (IIT) that was originally designed to fit the iCub robot (see [49]). In this case it has been mounted

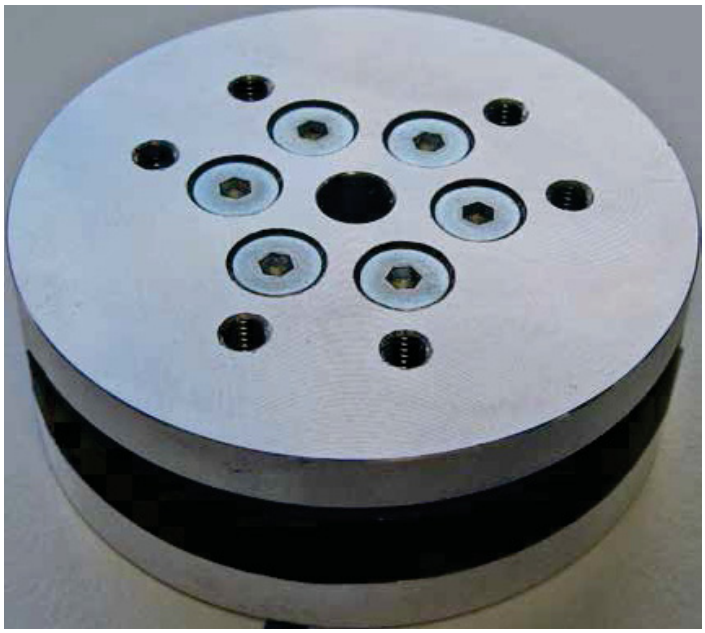

Fig. 6.3 FTSens force/torque sensor.

on the end-effector of the Puma 260 robot by means of an *ad-hoc* adapter. The specifications of the sensor are given below:

Dimensions $[\phi, H]$	45x18 mm
Weight	122 g
Power Supply	5 V ±10%, current consumption max 100 mA
Channels	Six, 3 torques (T_x, T_y, T_z) and 3 forces (F_x, F_y, F_z)
Measure range	2000 N (F_x, F_y, F_z) 40 Nm (T_x, T_y) 30 Nm (T_z)
Resolution	0.25 N (F_x, F_y, F_z) 0.049 Nm (T_x, T_y) 0.037 Nm (T_z)
Output data	16 bit, 6 channels, up to 1k messages/sec
A/D Converter	16 bit, 250ksps
Operating conditions	0 to 50°C, humidity <85% without condensation
Communication	CAN Bus 2.0B, 1Mbps

6.2 Developed Components

It follows a list of components developed for this particular setup in addition to the ones developed for the I-SUR project (see 4.5 and 5.4).

6.2.1 Puma 260 EtherCAT Master

This *Driver* component encapsulates the function provided by the SOEM library (see [48]). This library implements an EtherCAT master and allows to communicate with a set of EtherCAT slaves connected to an Ethernet port of the computer.

The *CANopen Over EtherCAT* (CoE) protocol has been exploited for the exchange of data with the drivers: the CoE protocol enables the use of the *CiA 402* device profile over EtherCAT. The CiA 402 device profile is internationally standardized as *IEC 61800-7-201* and *IEC 61800-7-301* and defines the functional behavior of controllers for servo driver, frequency inverters and stepper motors.

At each device is associated a finite state machine: the current state is represented by a *status-word*, the current command is represented with a *control-word*. The current state determines which commands are accepted and if high power is enabled.

When the component is configured, the available EtherCAT slaves are listed and for each driver it is added a relative service to the EtherCAT master component. Through these services it is possible to configure the slaves, to send commands to them and to monitor their state.

The component can be in the following states:

- **Not initialized.** The initial state, at the creation of the component;
- **Initialized.** The state of the component once all slaves have been configured;
- **Moving to nest.** The robot enters this state when a movement to the *nest* position is requested. Is is possible to request such a movement only while in the *ready* position;
- **Nest position.** The *nest* position is a mechanical constrained position used for the initialization of the encoders of the robot;
- **Moving to ready.** The robot enters this state when a movement to the *ready* position is requested.
- **Ready position.** The *ready* position is a special position used to enter or exit the *nest* position. From this state is possible to both go to the *nest* position or switch to the *operative* state;
- **Operative.** While in this state the robot is fully operative and it is ready to receive commands.

A final state machine representing the behavior of the component is shown in figure 6.4.

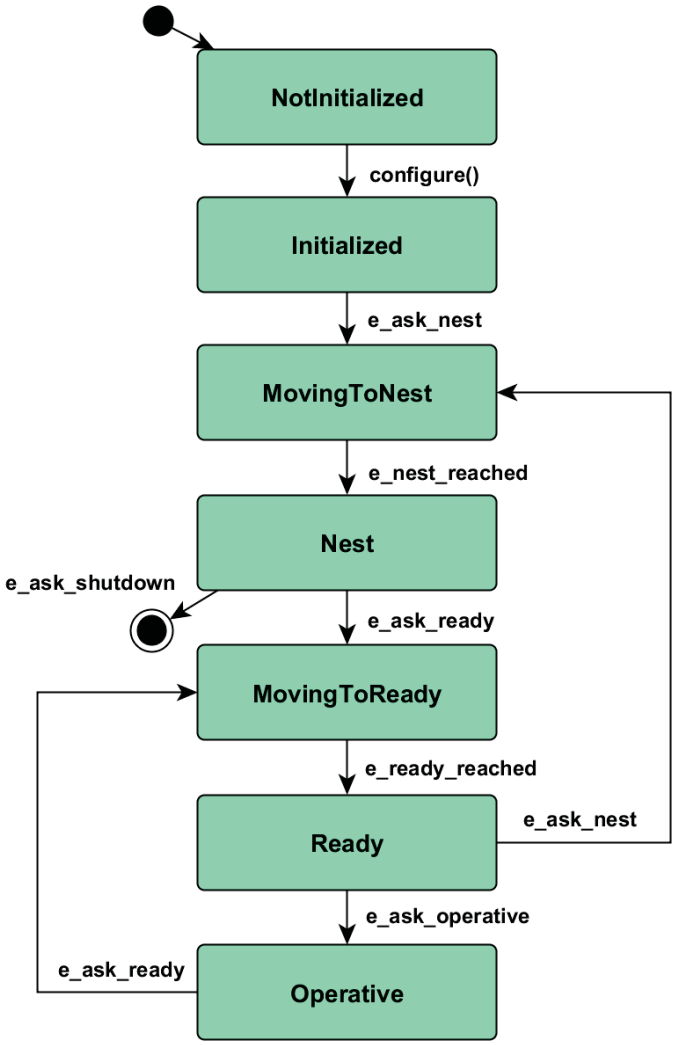

Fig. 6.4 Finite state machine for the EtherCAT master component.

Interface

The interface of the component is described as follows:

- **control_mode.** It selects the control mode adopted by the robot between position, velocity or torque mode;
- **slave_n.** A service that represents the nth slave connected to the master;
- **askNest().** A method used to request to the robot a predefined joints motion to the *nest* position;
- **askReady().** A method used to request to the robot a predefined joints motion to the *ready* position;
- **askOperative().** A method used to request to the robot the *operative* state;
- **desJointPosition.** An input port used to command a desired joint position;
- **desJointVelocity.** An input port used to command a desired joint velocity;
- **desJointTorque.** An input port used to command a desired joint torque;
- **measJointPosition.** An output port used to publish the measured joint position;
- **measJointVelocity.** An output port used to publish the measured joint velocity;
- **measJointTorque.** An output port used to publish the measured joint torque;
- **currentState.** An output port used to publish the current state of the robot;
- **events.** An output port used to publish events generated by the robot (e.g., execution of a command, failures, et al.).

6.2.2 FTSens Driver

This is a *Driver* component encapsulating the functions used for the communication with the FTSens force/torque sensor through CAN bus interface. Though this component it is possible to initialize the sensor, to configure it and to read the measured wrench.

6.2.3 Equivalent Wrench

Sometimes, it is useful to change the application point or the reference system of a wrench. For this purpose it has been created a *Calculation* component that encapsulates this function.

It the case of the Puma 260 setup this component is used to refer the measured wrench coming from the force/torque sensor to a particular point of the kinematic chain, typically the tip of the tool. To change both the reference system and the application point of a wrench, it is possible to use an *adjoin matrix* (see [50]) as follows:

$$F_c = Ad^T_{g_{bc}} F_b \tag{6.1}$$

where B and C are two coordinate frames and $g_{bc} = (p_{bc}, R_{bc})$ represents the configuration of frame C relative to B; F_b and F_c are representations of the same wrench in the reference systems B and C and the adjoin matrix $Ad^T_{g_{bc}}$ is defined as:

$$Ad^T_{g_{bc}} = \begin{bmatrix} R^T_{bc} & 0 \\ -R^T_{bc}\hat{p}_{bc} & R^T_{bc} \end{bmatrix} \tag{6.2}$$

where $\hat{p}$ is a skew-symmetric matrix obtained from a vector $p = [p_1, p_2, p_3]^T$ as follows:

$$\hat{p} = \begin{bmatrix} 0 & -p_3 & p_2 \\ p_3 & 0 & -p_1 \\ -p_2 & p_1 & 0 \end{bmatrix} \tag{6.3}$$

Interface

The interface of the component is described as follows:

- **change_application_point.** A property used to specify if the application point of the wrench must be changed;
- **change_reference_system.** A property used to specify if the reference system of the wrench must be changed;
- **referenceFrame.** An input port on which it can be provided a reference frame used to change the reference system and/or the application point of the measured wrench;
- **inWrench.** An input port on which it is received a wrench;
- **outWrench.** An output port on which it is published the input wrench with reference system and/or application point modified accordingly with the value of the reference frame;

6.2.4 Kinematics Solver

This is a *Calculation* component that encapsulates the forward and inverse kinematics solvers for a generic serial robot described through Denavit-Hartenberg parameters. It is implemented using the API provided by KDL and it can operate both on demand, by requiring a proper service, or through data flow.

In particular, when it is used for the resolution of the inverse kinematics problem, it is designed so that it publishes new data only when the solver returns a valid solution. This can depend on both the convergence of the solver within the iteration limit or the reachability of a desired Cartesian pose.

Since a closed form solution of the kinematics is available for the Puma 260 robot, it has been chosen to implement an alternative version of this component encapsulating this version of the solver. The system has been designed so that these two components are completely interchangeable and it is up to the user to decide which version must be deployed.

6.3 Deployment

It has been chosen to test the system on elementary task, in order to just being able to verify a proper deployment, configuration and coordination of the architecture. In particular, it is required from the robot to perform a simple motion primitive while keeping a compliant behavior. In the case of unexpected contact with the environment, the robot must try to follow the nominal trajectory while maintaining a stable behavior.

The set of component deployed is the following:

- **Puma 260 EtherCAT Master**;
- **FTSens Driver**;
- **Equivalent Wrench**;
- **Kinematics Solver**;
- **Multi-Arm Cartesian Trajectory Generator** (see 4.5.2);
- **Variable Admittance Control** (see 4.5.3);
- **Supervisor** (see 4.5.10);

The deployment of the components in shown in figure 6.5. About the configuration, given the simplicity of the task the same set of parameters have been kept during all its execution.

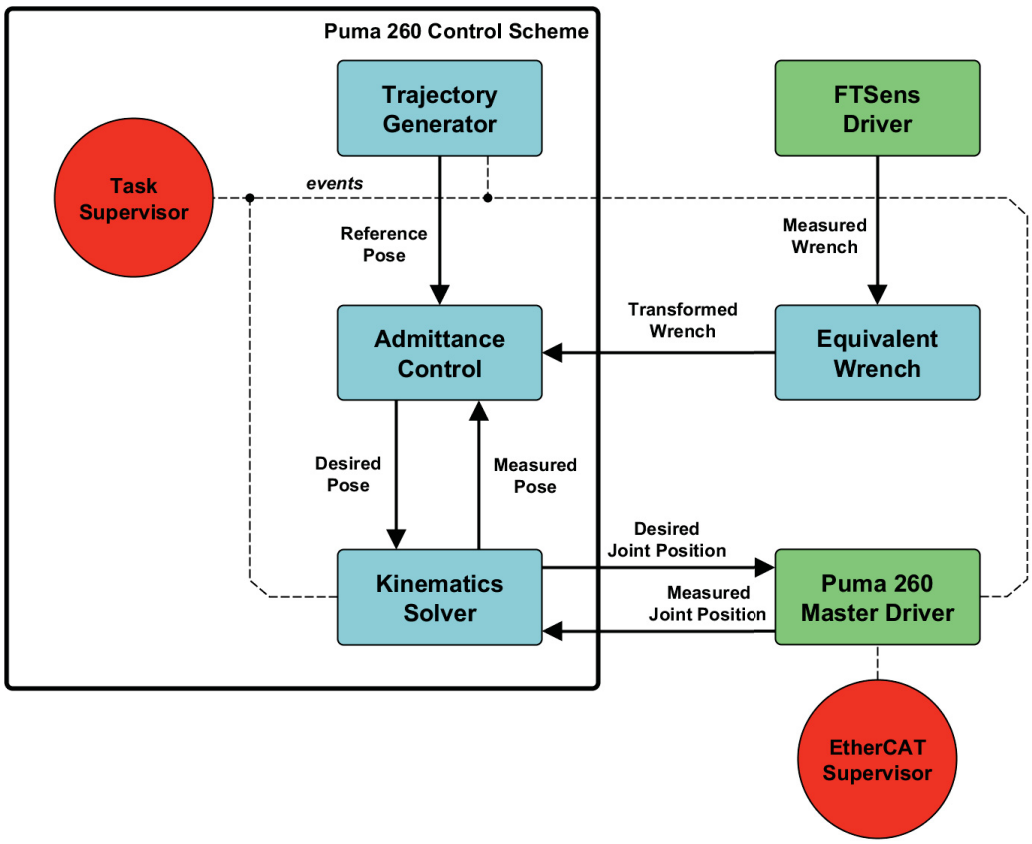

Fig. 6.5 Deployment of the control scheme used for the experiments with the retrofitted Puma 260.

6.4 Results

A simple contact tool has been fixed to the force sensor mounted on the end-effector of the robot and a curve surface has been placed inside the workspace, to simulate the interaction of the robot with an unexpected obstacle while following a desired trajectory.

The different phases of the task are represented in the frames shown in figure 6.6 and are described as follows:

- **A.** The robot starts from its *nest* position where it is initialized;
- **B.** From the *nest* position the robot moves to the *ready* position and goes in the *operative* state;
- **C.** The robot is moved to a default task position in which the admittance control is enabled;
- **D.** The robot reaches the starting pose of the trajectory;

- **E.** While following a nominal linear trajectory the robot touches an unexpected obstacle and the wrench measured by the force sensor is passed to the admittance control that ensures a compliant behavior;
- **F.** The motion continues and the robot remains in contact with the surface as soon as it is in collision with the desired trajectory;
- **G.** At this point the surface is not anymore on the path of the trajectory and the robot can go back to the nominal trajectory;
- **H.** The robot reaches the final pose of the trajectory.

In figure 6.7 the nominal trajectory passed to the admittance control is represented with a blue dotted line. The red line represents instead the actual trajectory followed by the robot due to the interaction. The letters along the path are associated to the correspondent frames of figure 6.6.

During this experiment it has been possible to effectively test the reusability of the control scheme previously adopted for the I-SUR project.

The admittance control has been reused without the need of modifications and the only operation required has been a tuning of the parameters of the control. This is perfectly acceptable because the *configuration* depends not only on the requirements of the task but also on the hardware available in the setup.

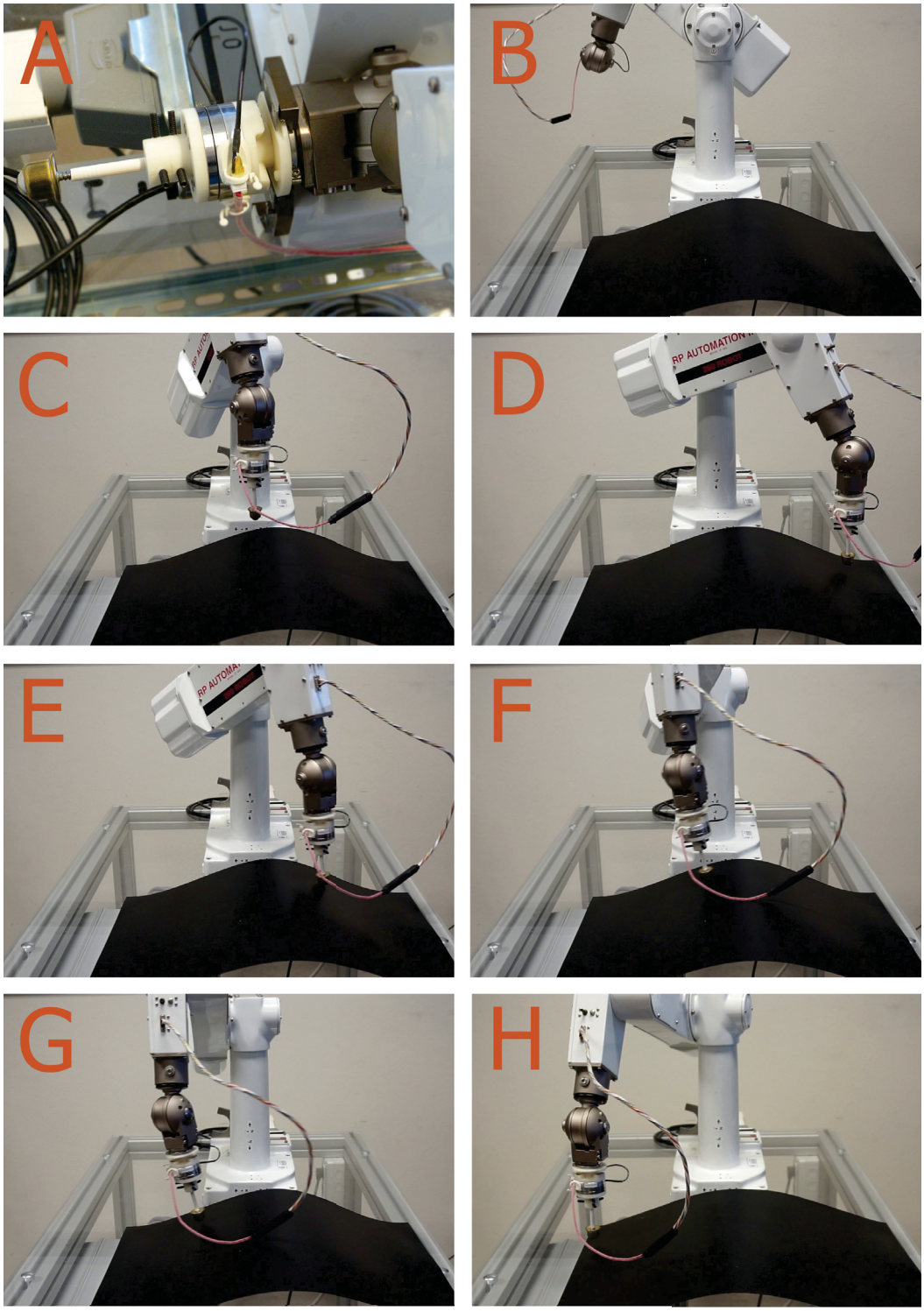

Fig. 6.6 Frames illustrating the task executed by the robot

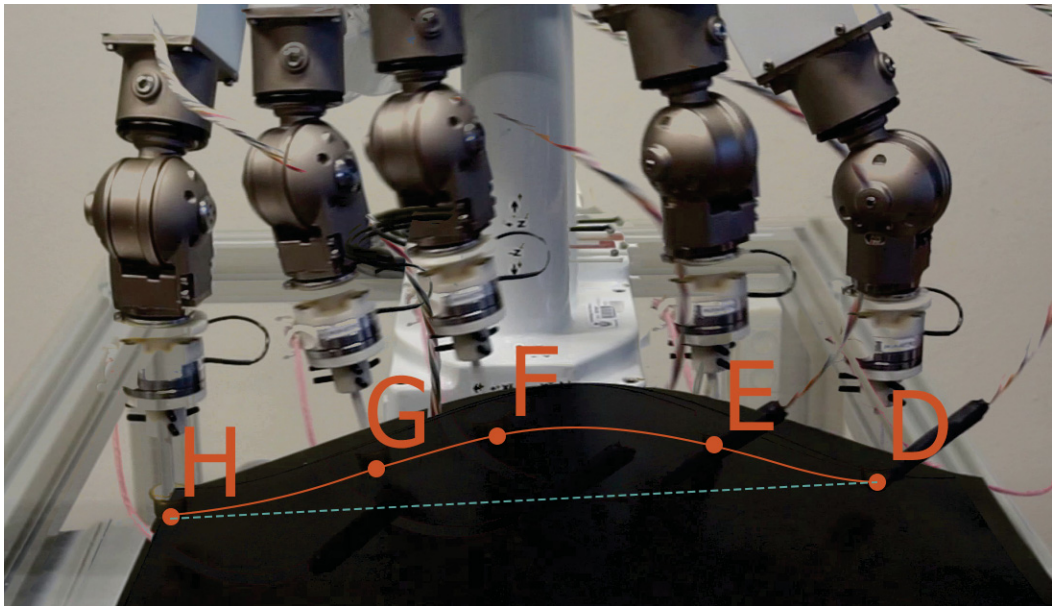

Fig. 6.7 Behavior of the robot while employing an admittance control

Conclusions

This thesis proposed a set of patterns for the development of a reusable component-based architecture for the robotics. These patterns have been defined aiming to maximize the properties of reusability, flexibility and modularity. The optimization of computational performances was not considered as an aspect that should prevail over the benefits of the mentioned properties, unless very critical situations are addressed.

With reference to the considered case studies, the application of the suggested *best practices* never leaded to problems in terms of the scheduling of the system. Anyway, at least for what concerns the OROCOS framework used for the implementation of the architecture, it has been observed that a deployment containing a large number of ports can introduce problems related to the memory. It follows that is up to the designer of the architecture to find the right compromise between the respect of the computational constraints imposed by an available platform and the achievement of a reusable system.

What it is important to underline is that the design of a reusable component-based system constitutes in general a rewarding approach under several aspects. For example, once a reusable component has been developed and tested, it become part of a set of available components and it can be employed for the implementation of several tasks. Moreover, the composition of components allows to describe complex functions starting from a set of available actions provided by the existing set of components. That means that within a context of applications, once a library of elementary components have been defined, it is possible to describe a task working mostly on deployment, configuration and coordination.

Several case studies have been introduced in which the suggested patterns have been employed, and thus it has been shown how it is possible to exploit them for the development of a complex application. These patterns have proven to support the process of implementation of a task, effectively reducing the development time and allowing a smooth integration of the system.

Future works aim to create a high-level framework for the robotics, composed by a set of components designed accordingly with the proposed patterns. The idea is to provide the user

with tools that enable a rapid prototyping of the control architecture, allowing to focus on the coordination and configuration of the system.

References

[1] C. Bergeles and G. Yang, "From passive tool holders to microsurgeons: Safer, smaller, smarter surgical robots," 2014.

[2] P. Gomes, "Surgical robotics: Reviewing the past, analysing the present, imagining the future," *Robotics and Computer-Integrated Manufacturing*, vol. 27, no. 2, pp. 261–266, 2011.

[3] W. Korb, R. Marmulla, J. Raczkowsky, J. Mühling, and S. Hassfeld, "Robots in the operating theatre –chances and challenges," *International journal of oral and maxillofacial surgery*, vol. 33, no. 8, pp. 721–732, 2004.

[4] J. Rosen, B. Hannaford, and R. M. Satava, *Surgical Robotics: Systems Applications and Visions*. Springer, 2011.

[5] V. Vitiello, S.-L. Lee, T. P. Cundy, and G.-Z. Yang, "Emerging robotic platforms for minimally invasive surgery," *Biomedical Engineering, IEEE Reviews in*, vol. 6, pp. 111–126, 2013.

[6] Intuitive Surgical, 2014. [online] http://www.intuitivesurgical.com.

[7] A. Tobergte, R. Konietschke, and G. Hirzinger, "Planning and Control of a Teleoperation System for Research in Minimally Invasive Robotic Surgery," in *Proceedings of the IEEE International Conference on Robotics and Automation*, (Kobe, Japan), pp. 4225–4232, May 2009.

[8] D. B. Camarillo, T. M. Krummel, and J. K. Salisbury Jr, "Robotic technology in surgery: past, present, and future," *The American Journal of Surgery*, vol. 188, no. 4, pp. 2–15, 2004.

[9] K. C. Curley, "An overview of the current state and uses of surgical robots," *Operative Techniques in General Surgery*, vol. 7, no. 4, pp. 155–164, 2005.

[10] P. Fiorini, R. Muradore, G. Akgun, D. E. Barkana, M. Bonfe', F. Boriero, G. De Rossi, R. Dodi, O. J. Elle, F. Ferraguti, L. Gasperotti, R. Gassert, K. Mathiassen, D. Handini, O. Lambercy, L. Li, M. Kruusmaa, A. O. Manurung, G. Meruzzi, H. Q. P. Nguyen, N. Preda, A. Ristolainen, A. Sanna, C. Secchi, and A. E. Yantac, "Development of a cognitive robotic system for simple surgical tasks," *International Journal of Advanced Robotic Systems*.

[11] M. D. McIlroy, "Mass-produced software components," *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[12] Microsoft, "Component Object Model." https://www.microsoft.com/com.

[13] Object Management Group, "CORBA (Common Object Request Broker Architecture) specifications." http://www.corba.org.

[14] Object Management Group, "Deployment and configuration of component-based distributed applications," April 2006.

[15] A. Shakhimardanov, H. Bruyninckx, K. Nilsson, and E.Prassler, "White paper: The use of reuse for designing and manufacturing robots," 2009. [online] http://www.robot-standards.eu//Documents_RoSta_wiki/whitepaper_reuse.pdf.

[16] D. Vanthienen, M. Klotzbücher, and H. Bruyninckx, "The 5c-based architectural composition pattern: lessons learned from re-developing the itasc framework for contraint-based robot programming," 2014.

[17] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2011.

[18] M. Klotzbücher, G. Biggs, and H. Bruyninckx, "Pure coordination using the coordinator–configurator pattern," *CoRR*, vol. abs/1303.0066, 2013.

[19] The Orocos Project, "Smarter control in robotics and automation." http://www.orocos.org.

[20] J. Arata, H. Kozuka, H. Kim, N. Takesue, B. Vladimirov, M. Sakaguchi, J. Tokuda, N. Hata, K. Chinzei, and H. Fujimoto, "Open core control software for surgical robots," *International Journal of Computer Assisted Radiology and Surgery*, vol. 5, pp. 211–220, May 2010.

[21] ROS, "An open-source Robot Operating System." http://www.ros.org.

[22] YARP, "Yet Another Robotic Platform." http://wiki.icub.org/yarp.

[23] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, pp. 231–274, June 1987.

[24] Object Management Group, "UML specifications." http://www.omg.org/spec/UML/2.3/Superstructure/PDF/.

[25] R. Ierusalimschy, L. H. de Figueiredo, L. Henrique, F. Waldemar, and W. C. Filho, "Lua - an extensible extension language," 1996.

[26] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo, "A constant-time dynamic storage allocator for real-time systems," *Real-Time Systems*, vol. 40, no. 2, pp. 149–179, 2008.

[27] *Hard real-time Control and Coordination of Robot Tasks using Lua*, (Czech Technical University, Prague), October 2011.

[28] L. Biagiotti and C. Melchiorri, *Trajectory Planning for Automatic Machines and Robots*. Springer Berlin Heidelberg, 2008.

[29] R. Dodi, F. Ferraguti, A. Ristolainen, C. Secchi, and A. Sanna, "Planning and simulation of percutaneous cryoablation," in *2nd AASRI Conference on Computational Intelligence and Bioinformatics*, vol. 6, p. 118–122, 2014.

[30] M. Bouri and R. Clavel, "The linear delta: Developments and applications," in *Proceedings of the International Symposium on Robotics and German Conference on Robotics*, (Munich, Germany), pp. 1198–1205, June 2010.

[31] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, pp. 72–82, December 2012. http://ompl.kavrakilab.org.

[32] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries.," in *ICRA*, pp. 3859–3866, IEEE, 2012.

[33] J. Kuffner and S. LaValle, "RRT-connect: an efficient approach to single-query path planning," in *Proc. of IEEE Int. Conf. on Robotics and Auomation*, pp. 995–1001, 2000.

[34] L. Villani and J. De Schutter, "Force control.," in *Springer Handbook of Robotics* (B. Siciliano and O. Khatib, eds.), ch. 7, Springer Berlin Heidelberg, 2008.

[35] C. Secchi, S. Stramigioli, and C. Fantuzzi, *Control of Interactive Robotic Interfaces: A Port-Hamiltonian Approach.*, vol. 29 of *Springer Tracts in Advanced Robotics*. Springer-Verlag, 2006.

[36] A. Franchi, C. Secchi, H. I. Son, H. H. Bülthoff, and P. Robuffo Giordano, "Bilateral teleoperation of groups of mobile robots with time-varying topology.," *IEEE Transactions on Robotics*, vol. 28, no. 5, pp. 1019–1033, 2012.

[37] M. Franken, S. Stramigioli, S. Misra, C. Secchi, and A. Macchelli, "Bilateral telemanipulation with time delays: A two-layer approach combining passivity and transparency.," *IEEE Transactions on Robotics*, vol. 27, no. 4, pp. 741–756, 2011.

[38] P. Robuffo Giordano, A. Franchi, C. Secchi, and H. H. Bülthoff, "A passivity-based decentralized strategy for generalized connectivity maintenance.," *International Journal of Robotics Research*, vol. 32, no. 3, pp. 299–323, 2013.

[39] C. Secchi, S. Stramigioli, and C. Fantuzzi, "Position drift compensation in port-hamiltonian based telemanipulation.," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4211–4216, 2006.

[40] D. Lee and K. Huang, "Passive-set-position-modulation framework for interactive robotic systems.," *IEEE Transactions on Robotics*, vol. 26, no. 2, pp. 354–369, 2010.

[41] F. Ferraguti, N. Preda, A. Manurung, M. Bonfe', O. Lambercy, R. Gassert, R. Muradore, P. Fiorini, and C. Secchi, "A tank-based interactive control architecture for autonomous and teleoperated robotic surgery," *IEEE Transactions on Robotics*, 2015. [Conditionally Accepted].

[42] E. Nuno, R. Ortega, N. Barabanov, and L. Basanez, "A globally stable PD controller for bilateral teleoperators.," *IEEE Transactions on Robotics*, vol. 24, no. 3, pp. 753–758, 2008.

[43] D. Lee and M. W. Spong, "Passive bilateral teleoperation with constant time delay.," *IEEE Transactions on Robotics*, vol. 22, no. 2, pp. 269–281, 2006.

[44] Khronos Group, "OpenGL: Open Graphic Library."

[45] Covidien. http://www.medtronic.com/us-en/index.html.

[46] Leap Motion Inc., "Leap motion." https://www.leapmotion.com, 2014.

[47] Elmo. http://www.elmomc.com.

[48] Simple Open EtherCAT Master. https://developer.berlios.de/projects/soem.

[49] IIT, "icub, an open source cognitive humanoid robotic platform." http://www.icub.org/.

[50] R. M. Murray, S. S. Sastry, and L. Zexiang, *A Mathematical Introduction to Robotic Manipulation*. Boca Raton, FL, USA: CRC Press, Inc., 1st ed., 1994.