TECHNOLOGY REPORT
published: 08 April 2016
doi: 10.3389/frobt.2016.00017

A Cross-Platform Tactile Capabilities Interface for Humanoid Robots

Jie Ma[1] *and Torbjørn S. Dahl*[2]*

[1] *China Research Laboratory, IBM, Beijing, China,* [2] *Centre for Robotics and Neural Systems, School of Computing, Electronics and Mathematics, Plymouth University, Plymouth, UK*

This article presents the core elements of a cross-platform tactile capabilities interface (TCI) for humanoid arms. The aim of the interface is to reduce the cost of developing humanoid robot capabilities by supporting reuse through cross-platform deployment. The article presents a comparative analysis of existing robot middleware frameworks as well as the technical details of the TCI framework that builds on the existing YARP platform. Currently, the TCI framework includes robot arm actuators with robot skin sensors. It presents such hardware in a platform-independent manner, making it possible to write robot control software that can be executed on different robots through the TCI frameworks. The TCI framework supports multiple humanoid platforms, and this article also presents a case study of a cross-platform implementation of a set of tactile protective withdrawal reflexes that have been realized on both the NAO and iCub humanoid robot platforms using the same high-level source code.

Keywords: tactile capabilities interface, humanoid robotics, robot skin, protective reflexes, robot software engineering

OPEN ACCESS

Edited by:
Nikolaus Vahrenkamp, Karlsruhe Institute of Technology, Germany

Reviewed by:
Fulvio Mastrogiovanni, University of Genoa, Italy
Kensuke Harada, National Institute of Advanced Industrial Science and Technology, Japan

****Correspondence:***
Torbjørn S. Dahl
torbjorn.dahl@plymouth.ac.uk

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *07 December 2015*
Accepted: *24 March 2016*
Published: *08 April 2016*

Citation:
Ma J and Dahl TS (2016) A Cross-Platform Tactile Capabilities Interface for Humanoid Robots. Front. Robot. AI 3:17. doi: 10.3389/frobt.2016.00017

1. INTRODUCTION

During the last few decades, robots have been used with success in various domains ranging from manufacturing (Merzouki et al., 2010), space exploration (Ambrose et al., 2010), and surgery (McMahan et al., 2011) to mining (Bednarz et al., 2011) and military assistance (Wooden et al., 2010). Developing robotic software is difficult and time-consuming, especially when the same functionality must be developed separately for robots with different physical dimensions, hardware control protocols, mechanical configurations, or actuators and sensors. Even on a single robot, it is common for low-level components, such as dynamics and servos, to vary due to upgrades during the robot's lifetime. The cost of robot software development can be reduced significantly if the software can be reused across different models and platforms. In general, the main challenges of developing humanoid robot software are *modeling complexity*, *modularity*, and *repeatability*.

Humanoid robots are often equipped with a large number of actuators and sensors. The first difficulty a developer may encounter is to learn these specifications. Even when building a simple robotic behavior with just handful devices, nevertheless, it may be time consuming for the developer to work out the correct mappings from the platform-related infrastructure to build an appropriate behavioral model. For example, a humanoid robot NAO that our research employed has 21 DoF servos and 648 tactile sensors (see **Figure 1**), but a withdrawal reflex behavior studied in §4 is only interested in 5 servos and about 30 taxels. To identify and configure the correct taxels may become a challenge for the robotic behavior engineer.

From the perspective of the software engineering, modularity is also important in humanoid robot. Currently, humanoid robot projects are usually requiring intensive collaboration among

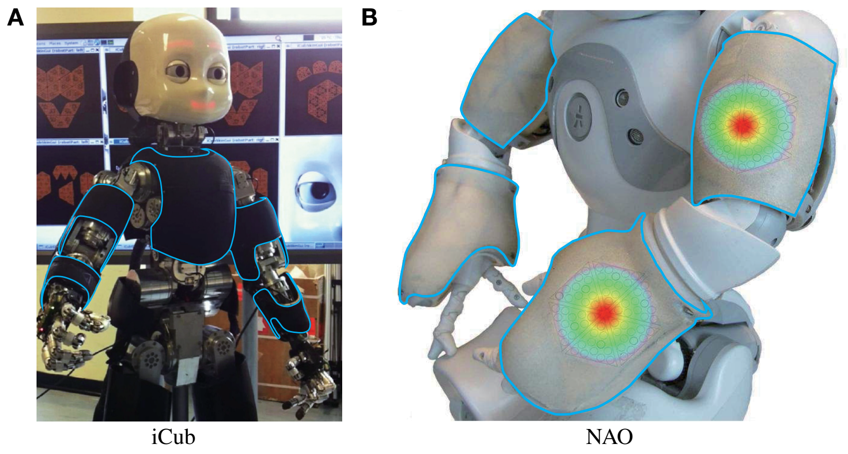

FIGURE 1 | (A,B) illustrate iCub and NAO robots, respectively. Their equipped robotic skin is highlighted. The TCI framework presented in this article was evaluated on both platforms and both on the real robots and in simulations.

different specialists. Developers may use diverse programming languages, operating systems, or even computing hardware. Thus, the need to separate functions into reusable modules has been increasingly growing, so people can just focus on smaller-scaled problems. Without a cross-platform interface, however, it makes communication and integration difficult as it typically triggers extra work to facilitate the interaction between the separate components and thus makes modularity difficult to achieve. Shared generic interfaces can make robotic software more extensible and reduce the couplings among modules. They can also facilitate the development by supporting multiple operating systems and programming languages.

After the development of a behavior for a specific robot, software engineers are commonly interested in transferring the same behavior to other types of robot. As a matter of fact, behavioral transfer is not easy to achieve and the repeatability of humanoid robotics is sometimes criticized for being difficult to reproduce outside of their original laboratories (Anderson and Thomaz, 2010). The main reason making repeatability difficult is the hardware differences among the humanoid robots. The cost of directly migrating a platform-specific solution to a new type of robot is high without a decent generic interface, because a developer needs to figure out the geometric transformations and adjust high-level behavioral parameters correspondingly. By doing this, the repeatability of the behavior is broken and so it becomes obscure to verify and compare the effectiveness of the same behavior on the new robot.

The work presented in this article goes beyond traditional robot middleware platform by attempting to hide all platform-specific details from developers and, thus, allow them to produce reusable cross-platform behaviors via an abstract interface. The interface focuses on interpreting abstract information for different native different robot arms with different physiologies. This article presents results in developing a cross-platform tactile capabilities interface (TCI) that aims to improve the reusability of humanoid robot software and hardware. The results presented are limited to humanoid arms but includes a standardization of both actuators and tactile sensors that covers a large area of a robotic arm.

The research presented is directly motivated by our experiences from developing cross-platform software during the FP7 ROBOSKIN project (Cannata et al., 2012). Our research involved different humanoid robots, including the NAO and iCub robots (see **Figure 1**). During the research, the main challenges were to develop generic robot capabilities. In particular, we developed prototype algorithms for one robot platform and later re-implemented them on others. The objective of TCI is to facilitate such transfers of robot capabilities by providing a generic interface that is practical for a range of different humanoid robots. Our approach aims to enable the developers working on cross-platform capabilities, in particular protective withdrawal reflexes, to focus on controlling an abstract, platform-independent robotic component through a set of abstracted interfaces. TCI consists of generic actuator interfaces and generic robot skin interfaces. It acts as an interpreter translating the messages between the cross-platform algorithms and the platform-specific layers of the actual robots. High-level algorithms then become reusable and extensible, and new robots can be supported by providing them with support for TCI.

In §2, related concepts, systems, and literature are reviewed. The theories and methods for promoting robot software reusability are discussed, and a selection of robot middleware platforms compared. §3 presents the architecture implementing our interface and discusses its design. The TCI specifics of the arm actuators and the arm skin sensors are presented in §3.1 and §3.2, respectively. This is followed by a case study of how TCI was used to support a cross-platform implementation of a humanoid robot protective arm withdrawal reflexes in §4. Conclusions are made in § 5.

2. RELATED WORK

A humanoid robot system commonly consisted of a set of layered modules. Low-level modules focus on solving hardware-related preprocessing and reasoning problems, such as localization and sensory data fusion, whereas high-level ones are making cognitive behavioral decisions based on the high-level states produced by mother modules. To develop a generic software for controlling multiple type of humanoid robots is usually difficult, and the key problem is to design a high-level robot behavior that can be easily deployed on different robots.

Machine learning (ML) algorithms can be one possible solution, which provides a generic way of giving different robots the same capabilities, even in the absence of a detailed understanding of the underlying specifications or kinematics model. The core philosophy of ML is data driven; it focuses on the mappings from a robotic action to the corresponding feedback. Instead of directly solving transformation problems, ML undertakes a training process to understand the outcomes of actions. During the training, it continuingly changes action with an aim of achieving a certain behavior on a robot. Such approaches include learning the kinematics and dynamics of a generic robotic arm (Atkeson, 1989; Caligiore et al., 2010), generic trajectory tracking using neural control (Martins et al., 2008), and high-level humanoid behaviors such as learning biped locomotion (Huang et al., 2001; Ma and Cameron, 2009a,b, 2011).

The training processes of the ML approaches can be time consuming both in terms of acquiring and processing training data. The learning results may also be unreliable without performance guarantees, as sometimes the results are represented in the form of an implicit neural network, making users difficult to verify behavioral effectiveness. However, in reality, robot developers are not completely accessible to the specifications of a new robot. Instead, they are commonly interested in model-driven approaches that directly translate actuator and sensor data to a new platform. More importantly, the translation should be better processed in real time without waiting for the learning phase to complete. ML approaches, therefore, have disadvantages in real-time data processing. Compared with ML approaches, even hard-coded solutions can be implemented quicker and can sometimes provide reliable performance across a problem space.

In order to improve the reusability of hard-coded transformations, we have proposed the cross-platform tactile capabilities interface (TCI). This is a model-driven approach consists of a set of standardized representations and functions for humanoid actuators and tactile sensors. We have realized the interface for the humanoid arms of iCub and NAO robots.

This section reviews related approaches to increase the reusability of robot software and hardware. Currently, there are several popular robot middleware platforms, and their features are discussed and compared in §2.2. Since TCI provides a generic platform not only for tactile sensors but also for the actuators using them, this section also includes the kinematics features of the middleware. TCI uses YARP (Yet Another Robot Platform) (Metta et al., 2006) as its middleware platform because it was already been supported on several humanoid robots. YARP also supports a wide range of networking protocols that can be flexibly deployed in diverse circumstances. Technical details of YARP will be further discussed in this section.

2.1. The Reusability of Robotic Software

Robot software reusability can be achieved by decomposing robot functions into modules connected to each other within a shared middleware framework. A middleware framework typically provides a fundamental infrastructure for module interaction including abstractions for sensors and actuators, so that different abstract modules are not restricted to specific robot hardware or operating systems, but can be instantiated by multiple specific solutions. With the help of such middleware, supplemental tools, such as behavioral abstraction composition applications, can also be provided in order to promote reusability further.

2.1.1. Modularity

In software engineering, modularity implies that software is broken down into a number of simpler modules. High modularity means more modules and less coupling, i.e., dependency among them. At the other end of the spectrum, monolithic approaches implement all the required tasks within a single program. Designing reusable software involves finding the best trade-off between too much modularity, which can be wasteful, and too little, which limits reusability (Brugali and Scandurra, 2009). A monolithic approach must include all the aspects of robot control, from low-level messaging to high-level algorithms. This approach includes solving platform-specific issues and, as a consequence, reusability is difficult to achieve. On the other hand, too much modularity can also reduce reusability. Practically, it is time consuming to integrate a large number of modules, and maintaining and documenting many modules also demands a large amount of resources. With the balanced level of modularity, the high-level modules can focus on cross-platform capabilities using abstract interfaces. The low-level modules are then responsible for translating the abstract representations and algorithms to platform-specific data and instructions.

2.1.2. Middleware and Toolkits

In order to promote modularity, much effort has been made to create shared robot middleware for different robots. Middleware platforms typically provide a communications framework for robot software modules, supporting both low-level control and high-level algorithms. Middleware typically also aims to reduce infrastructure-level programming and promote the development of reusable robot modules. This can be done through standardized interfaces, drivers for diverse hardware, and supporting multiple programming languages. The last point is often achieved by basing module communication on cross-platform communication technologies, such as TCP/IP sockets, supported by most programming languages.

Standardized data representations and control and communication interfaces allow middleware platforms to also provide a number of development toolkits for speeding up development and debugging. These toolkits can comprise visualization tools to display data, such as *playercam* (from *Player*) that remotely monitors camera images, a graphical user interface (GUI) that manages the modular processes and their connections, such as *yarpmanager* (from *YARP*), and a utility to query and inspect code trees and find dependencies, such as *rospack* (from *ROS*). Abstraction mechanisms, such as *Rosbridge* (Crick et al., 2012), can further increase reusability by allowing third-party tools, e.g., the image processing library (IPL) and the OpenCV library to be accessed from within reusable robot modules. A detailed analysis of the specific features of a selection of popular middleware platforms is presented in §2.2.

2.1.3. Behavior Abstraction and API Standardization

Another mechanism that increases the reusability of robot software is behavior abstraction. Complex robotic tasks are easier to achieve if they can be decomposed into a hierarchy of capabilities or behaviors. In cases where such decomposition is possible, subbehaviors constitute potential reusable modules, leaving high-level deliberative decision-making mechanism, such as planning to focus on scheduling abstract behaviors, rather than handling a higher number of low-level actions directly. Such behavioral decomposition has been widely used in various AI systems (Maes, 1991; Stone and McAllester, 2001; Nesnas et al., 2006), especially multi-agent systems (MAS) where cooperative behaviors are needed (Stone, 2000; Ma, 2011). It has also been successful in learning scenarios.

Abstract robot behaviors need to present standard data structures and application programming interfaces (APIs) in order to be used by high-level decision processes. The task description language (TDL)[1] is a C++ library that provides syntactic support for behavioral decomposition, synchronization, and execution monitoring. TDL reduces the difficulties of maintaining task-level behaviors, and it has been successfully used in several mobile robot projects, including CLARAty (Nesnas et al., 2006), a reusable platform for NASA's robots. Another similar example is the humanoid motion planner that is designed for humanoid robots in the Joint French-Japanese Robotics Lab (JRL) (Yoshida et al., 2005). The motion planner uses hierarchical architecture to control multiple reusable dynamic tasks such as path planning, gait generation, and collision checking.

2.2. Robot Middleware

Robot middleware platforms provide abstract platforms for robot software. They promote software modularity by providing tools that support flexible communication between different robot components, including communication between distributed processes. Most robot middleware uses networking packages to connect modules, making modules platform independent. Allowing distributed modules also means that modules can be executed on different processors, potentially under different operating systems and stored on diverse media.

YARP (Yet Another Robot Platform)[2] is a robot middleware platform designed for humanoid robots. It is a lightweight open-source platform derived from University of Genova and MIT, and it supports many mainstream programming languages, such as C++, Matlab, Python, JAVA, Perl, and L. It connects modules using various protocols, such as TCP, UDP, UDP multicast, and HTTP. A YARP-based system is a peer-to-peer network of *port* objects, where each object has *read* and *write* ports to receive and send data streams, respectively. One of the advantages of YARP is its synchronization mechanism. A write port can choose whether to wait for all its read ports before or after each update step. From the perspective of a developer of high-level behaviors, a kinematic chain, such as a robotic arm or leg, is implemented using a *PolyDriver* class. Modules based on this class can be read from and written to using an ordered vector of values based on the kinematic joint angles. The low-level limits of an actuator, however, are not effectively managed by the framework, e.g., YARP does not check the velocity limits of a servo before sending a command. It may even break the servo.

ROS (Robot Operating System)[3] takes slightly different messaging approach. Instead of using *read* and *write* ports it employs a publish–subscribe mechanism (Quigley et al., 2009). *Nodes* are computational processes, which communicate with each other by passing messages. A node sends messages by publishing it to a given *topic*, and nodes subscribe to selected topics to receive messages. ROS is also written in C++ but supports other programming languages as well, including Python, Octave, and LISP. In terms of communication, ROS supports the TCP and UDP protocols. In ROS, a kinematic chain is presented as an actuator array, where actuator properties are also defined such as velocity and torque limits. Actuator channels can be subscribed separately or manipulated at the same time by sending vectors of joint angles in a particular order.

The OROCOS (Open Robot Control Software) project (Bruyninckx, 2001; Bruyninckx et al., 2003) is different from YARP and ROS in that it does not emphasize communication between robot components. Instead, it focuses on toolkit libraries for solving common problems encountered by industrial robots. OROCOS is designed for a single robot, and it is not suitable for multi-robot systems where different robots need to cooperate with each other (Namoshe et al., 2008). Based on the GPL license, OROCOS is composed of four C++ libraries: a Kinematics and Dynamics Library (KDL) that solves real-time kinematics and dynamics problems, a Bayesian Filtering Library (BFL) that provides generic filtering functions such as Kalman Filter and Particle Filter, and two supporting libraries that couple robotic components with each other using debugging tools, i.e., Component Library (OCL) and Real-Time Toolkit (RTT). Originally, OROCOS only supported C++, but libraries have been integrated to connect OROCOS to other robotic middleware such as YARP and ROS. Through this mechanism, other languages are indirectly supported. The advantage of the OROCOS libraries is that they implement dynamics algorithms independently of platforms using a predefined abstraction and formalization of the underlying platform hardware, such as the kinematics.

Other robot middleware platforms include the Player/Stage Project[4] and LCM (Lightweight Communications and Marshaling).[5] The Player/Stage Project (Gerkey et al., 2003) consists of *Player*, a robot device server with standardized interface to sensors and actuators, and *Stage*, a 2D multi-robot simulator. Player provides a network interface to a variety of robot and sensor hardware, and many Player-based applications have been adapted for use under ROS. Player can be regarded as a cut-down version of YARP in that it only supports reliable TCP protocols without advanced synchronizing mechanism. In particular, in Player,

[1]TDL is available at http://www.cs.cmu.edu/~tdl/
[2]YARP is available at http://eris.liralab.it/yarp
[3]ROS is available at http://www.ros.org
[4]Player/Stage project is available at http://playerstage.sourceforge.net
[5]LCM is available at http://lcm-proj.github.io/

device interfaces are not buffered. Compared to Player, LCM is a relatively new library that aims to provide a low-latency message passing system for real-time robotics applications. Comparing to ROS, LCM showed notably better transmission efficiency by leveraging its UDP multicast infrastructure (Huang et al., 2010). With regards to representing a kinematic chain, the original *Player* does not support chained actuators. Instead, each actuator has to be used on its own, and it is up to the client application to keep track of any kinematic relations. Similarly, LCM is a lightweight platform for sharing information efficiently. It also does not natively support the management of kinematic chains.

Table 1 compares the properties of different robot middleware platforms in terms of network protocols, communication mechanisms, and open-source licenses. Although the number of the robotic devices in YARP is not as great as in ROS, YARP supports a greater number of humanoid robots. Player is a popular platform for wheeled robots, especially in the navigation domain. However, it is rarely used for humanoid robot control. Therefore, the work presented in this article has adopted YARP as the middleware platform due to its better support for humanoid robots and network communication.

The aforementioned frameworks provide general-purpose mechanisms to design and implement concurrent and distributed software robot components. However, in the particular domain of robot skin area, these frameworks may have problem in processing tactile sensory data. Robot skin often contains a large scale of taxels, which will cause latency for the frameworks to handle a huge volume of data. To solve this problem, Skinware, a framework designed for the large-scale skin, was proposed recently (Youssefi et al., 2014, 2015a,b). Skinware especially emphasizes on real-time tactile data processing and minimizes the latency for concurrent queries. It provides a unified interface to access information originating from heterogeneous robot skin systems and assures portability among different robot skin solutions.

Different from the Skinware, TCI proposed in this article focuses on solving geometrical transformations of the taxels and providing online actuator translations for tactile-based behaviors.

TABLE 1 | A comparison of popular robot middleware platforms, including the supported humanoid robot platforms, communication protocols, and interaction models.

Platform	Humanoid robots	Protocols	Model	License
YARP	NAO, iCub, Babybot, Obrero, Domo, COG, Kismet, and BERT2 KASPAR	TCP, UDP, UDP multicast HTTP, and QNet	R/W	LGPL
ROS	NAO, Romeo, Reddy, and Kondo KHR	TCP and UDP	P/S	BSD
OROCOS	Robonaut	TCP	C/S ORB SRB	GPL
Player	–	TCP	R/W	GPL
LCM	–	UDP multicast	P/S	LGPL

The license under which they are published is also included.
R/W, read/write; P/S, publish/subscribe; C/S, client/server; ORB, object request broker; SRB, service request broker.

Real-time data processing and latency analysis is not the focus of our work. Currently, even with the help of the aforementioned robotic middleware, it is still difficult to directly represent a generic kinematic chain of an arm for multiple humanoid robots. This is due to the fact that different robots use different reference systems for angles and speeds and also due to their different kinematics. **Listing 1** demonstrates the different code implementing the same reflex motion (see **Figures 6C,D**) on the iCub (Gamez et al., 2012) and NAO humanoid robots through YARP. For the right arm, NAO has 6 DoF while iCub has 16. The iCub robot also has more low-level limits, e.g., for damping and speed. The two extracts of code introduce difficulties for maintainability and reusability because each robot arm has a unique initial position tuple and unique coordinate systems. These differences introduce extra costs when it comes to supporting multiple humanoid platforms, even for simple generic motions. This problem is addressed by the TCI, which forces arms to be represented in the form of abstracted five DoF, disregarding the DoF it actually has. The interface is presented in detail in this article.

3. THE CROSS-PLATFORM TACTILE CAPABILITIES INTERFACE

One way of reducing the cost of developing cross-platform humanoid behaviors is to provide a generic interface that allows developers to reuse the same code for different robots. Since the robot control process is bidirectional, a generic robotic interface at least consists of two functions: incoming *sensor abstraction* and outgoing *actuator abstraction*. The differences in robot control software for various robots platforms stem from hardware issues, such as different physical dimension and mechanical configuration, as well as from software issues, such as platform-specific speed and angle units and servo-indexing mechanism.

Different humanoid platforms also have different schemas for joint indexing, as can be seen from the code provided in **Listing 1**. When migrating high-level code, it is up to the developer to figure out how to transfer abstract motions to the target robot. The problem is that this transfer is robot specific and, as a result, a different transfer is required for each robot that is to be supported, accommodating their unique joint indexing or naming. TCI provides a generic representation of humanoid robot arms, including joint control, joint position data, and kinematic chain information and data from robot skin sensors covering large areas of the arm. Such standardization promotes robotic reusability by hiding the low-level differences between specific robot platforms. In order to implement the interface, other platforms must be represented in a way that conforms to the specified data and command formats, e.g., a robot using log-encoded joints must provide a translation layer to convert the generic interface commands into platform-specific commands. Similarly, it must provide a translation layer converting the platform-specific data format to the format used by our interface.

In TCI, the DoF of a generic arm is fixed. All arms are represented by 5 DoF with indices from 0 to 4, always referring to the shoulder pitch/roll/yaw and the elbow pitch/yaw. Our interface enforces such a referencing standard even when the underlying

LISTING 1 | YARP-based code for NAO and iCub implementations of a single reflex motion.

```
/**** NAO robot ****/
const int DOF = 6;
double StiffnessArr[DOF]={0.8, 0.8, 0.8, 0.8, 0.8, 0.8};
double ReflexArr[DOF]={1.19, −1.10, 2.07, 0.04, 0.0, 0.0};
double MIN[DOF]={−2.09, −0.31, −2.09, −1.54, −1.82, 0.0};
double MAX[DOF]={2.09, 1.33, 2.09, −0.03, 1.82, 1.00};
pos− >SetStiffness(StiffnessArr);
for (int i=0; i<DOF; i++){
  StiffnessArr[i]=CheckLimits(MIN[DOF], ReflexArr[i], MAX[DOF]);
}
pos− >positionMove(ReflexArr);
/**** iCub robot ****/
const int DOF = 16;
double StiffnessArr[DOF]={0.4, 0.4, 1, 0.2, 0.2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
double DampingArr[DOF]={0.03, 0.03, 0, 0.01, 0.2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
double SpeedArr[DOF]={20, 20, 0, 20, 40, 0, 0, 50, 50, 50, 50, 50, 50, 50, 50, 50};
double ReflexArr[DOF]={−16.1, 83.3, 0, 15.5, −59, −0.45, 0, 0.3, 50, 24, 90, 5, 90, 16, 90, 115};
double MIN[DOF]={−90, 15, −30, 15, −90, −90, −20, 0, 10, 0, 0, 0, 0, 0, 0, 0};
double MAX[DOF]={10, 95, 50, 105, 90, 0, 40, 60, 90, 90, 180, 90, 180, 90, 180, 270};
SetStiffness(StiffnessArr);
SetDamping(DampingArr);
SetSpeed(SpeedArr);
for (int i=0; i<DOF; i++){
  StiffnessArr[i]=CheckLimits(MIN[DOF], ReflexArr[i], MAX[DOF]);
}
pos− >positionMove(ReflexArr);
```

platform is missing a particular DoF, e.g., no elbow yaw, or has extra DoF. Our interface also standardized the kinematic chain information, using a fixed grounding point, fixed units, ranges, and signs to represent angles and distances.

For the skin sensor data, rather than providing information related to indexed touch-sensitive transistors known as "taxels," our interface uses a spatial reference framework where a taxel is represented as a point in space, defined relatively to the underlying kinematics. Our implementations of the TCI translate the messages between the abstract high-level representations and the specifics of the individual robot representations in real time. This frees a developer from the tedious task of repeatedly resolving the low-level configuration differences.

In practice, developers need an interface to be flexible. Generic capabilities that can be potentially migrated to different robots can be implemented using our abstract interface. However, it is common to mix the generic capabilities with platform-specific capabilities making use of platform-specific sensors and actuators. In our layered architecture, presented in **Figure 2**, the platform-specific API is available through YARP. The complete architecture consists of four layers. Layer L1 is the low-level robotic controller layer. This is the lowest layer that a module can access and contains the hardware specific APIs. Layer L2 is the YARP middleware layer, which provides the communicative platform for modules to send commands and transfer data. This layer also includes the YARP converters that connect specific platforms to the YARP framework, e.g., NaoYarp[6] is a YARP interface for the NAO humanoid platform that wraps up the official NaoQi interface using YARP ports. Our abstract humanoid arm interface forms layer L3, which contains two core interfaces, the actuator interface, i.e., the arm

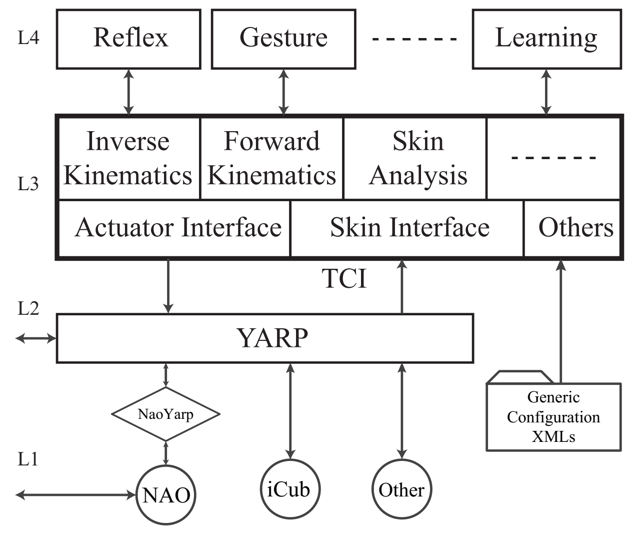

FIGURE 2 | The layered architecture of TCI. L1 and L2 are the low-level robotic controller layers and the YARP middleware layer, respectively. The tactile capabilities interface is defined in L3, which is followed by L4 that consists of reusable cross-platform capabilities.

kinematic chain, and the skin interface. These interfaces are presented in detail in §3.1 and §3.2, respectively. Our architecture also provides platform-specific information and calculations through modules, such as forward kinematics (FK) and inverse kinematics (IK), and technologies that have been extensively discussed by Diaz-Calderon et al. (2006). The run-time availability of platform-specific data, such as the kinematics, would allow the integration of our abstract interface and generic libraries such as OROCOS. We aim to include a querying mechanism in future versions of

[6]The NaoYarp software can be found at https://github.com/cbm/NaoYARP

our interface in order to enable the use of third-party libraries in the implementation of platform-independent capabilities. In the top layer, L4, high-level capabilities use our abstract interface to achieve generic tasks, such as withdrawal reflexes, gesture reproduction, and other robotic learning activities.

3.1. The Generic Actuator Interface

The configuration of the actuators of humanoid robots varies dramatically even when they are located on the same kinematic chain of a humanoid part. Practically, each actuator has some unique properties such as speed limits, position limits, initial position, axes, and stiffness settings.

A humanoid robot is usually designed with a specific domain of research in mind, e.g., the iCub robot supports research on cognitive representations for control of the robot head, arms, and hands. On the other hand, NAO robots are designed for locomotive behaviors in research domains such as robot football. As a result, corresponding robot parts on different platforms often have different features, including different degrees of freedom (DoF) and different sequencing of the individual DOF on a kinematic chain. The robot *head* for example has six DoF on the iCub but only two on the NAO. Similarly, the robot *arm* has sixteen DoF on the iCub but only six on the NAO.

Currently, our implementation relies on XML to specify the platform-specific elements of the actuator and skin interfaces. The generic actuator interface consists of a vector of abstract robot parts. Each part is accessible to the high-level components at runtime, providing platform-specific information in a standardized generic format. This unified interface reduces the development complexity for high-level modules by allowing them to focus on the key actuators and ignore any redundant ones that are not needed within a module. In the XML configuration file, the platform-specific information must be presented in terms of the defined number of actuators, axes, and initial positions. As our first attempt, each generic actuator is considered as a linear transformation, e.g., a specific abstract actuator, *GS* maps to a specific destination servo *DS*. The transformation can be defined as in equation (1). Nevertheless, our interface is not limited to this transformation. Other controls, such as transformation in Cartesian space, can also be configured using XML. However, the implementation of other kinematics models will be further evaluated in our future work.

$$GS = \gamma DS + \sigma \tag{1}$$

In equation (1), γ is the transform factor and σ is the transform displacement. Thus, a complex robot part can be generically defined as a vector of generic actuators, each of which is a transformation to a specific destination servo. This representation is formalized in equation (2).

$$GenericPart = \left\langle \, GS_i|_{i=1}^{n} \right\rangle = \left\langle \, \left(\gamma_i DS_{mapping(i)} + \sigma_i\right)\Big|_{i=1}^{n} \right\rangle \tag{2}$$

This is the configurations of a vector of servos, and the other properties of a kinematic chain, such as length information, are defined elsewhere. The difference of the limb lengths also affects the position of the end-effector position for given angles.

LISTING 2 | An example of the XML configuration file of a generic robot part. It defines an abstracted *head* part with 2 generic servos mapping from the robot part called *icub_head*. The generic servo 1 is transformed from the destination servo 2 with predefined transform properties and servo limits. The initial position of each generic servo vector is also configured.

```
<Generic_Robot_Part>icub_head</Generic_Robot_Part>
          <generic_servo_number>2</generic_servo_number>
          <generic_servo id="1">
             <destination_servo_id>2</destination_servo_id>
             <transform_factor>1.00</transform_factor>
             <transform_displacement>0.00</transform_displacement>
             <stiffness>1.00</stiffness>
             <damping>0.00</damping>
             <speed>20.00</speed>
          </generic_servo>
             ......
<initial_positions>
             <servo type="deg" destination_servo_id>="0" pos="-2.00"/>
             <servo type="deg" destination_servo_id>="1" pos="15.00"/>
</initial_postions>
```

For position-based behaviors, inverse kinematics and forward kinematics modules (see **Figure 2**) are used to calculate the low-level angles. The servos that make up one generic robot part are not necessarily from the same destination robot part; this feature gives users more flexibility to design robot parts properly. An example of the XML configuration file of a generic robot can be found in **Listing 2**. The generic servo *1* is transformed from the destination servo *2* with $\gamma = 1$ and $\sigma = 0$.

The XML configuration file contains not only the underlying data needed for controlling the real servos but also the information necessary to support a run-time reconstruction of the kinematic chain, e.g., TCI can provide a YARP device that gives run-time access to the Denavit–Hartenberg parameters as well as servo-specific information such as angle and velocity limits. Such a device facilitates the use of generic code that uses third-party libraries for generic computations such as inverse kinematics (IK) calculations (Diaz-Calderon et al., 2006).

3.2. The Generic Robot Skin Interface

A robotic skin sensor is a tactile sensor that gives a robot the "sense of touch" over large areas of its surface. Recently, Dahiya et al. (2010) have extensively compared more than 30 robotic tactile sensors in terms of their transduction method, number of taxels, range of force, and force sensitivity. Though there is no standardized representation of robot skin sensors, they can be regarded as a set of taxels (tactile pixels), where each taxel is located on the same continuous surface and each taxel is able to report, in real time, the force of any contacts. Our interface assumes that the different skin sensors represented have the same level of sensitivity. If this is the case, their readings may be normalized in interface implementation layer, L3, in **Figure 2**. Our interface, as presented here, is not yet sophisticated enough to cover skin sensors with different sensitivity.

One of the main difficulties in creating a generic interface for robot skin sensors has been the identification of a generic representation of a geometric model of the surface on which the taxels are located. Without such a model, it is difficult to reconstruct accurately and reliably the relative position and proximity of taxels across the surface of the robot. Although other information, such as the number of taxels, may also be useful, in practice, we have found that many high-level behaviors normally do not require such information and so we can achieve an acceptable performance level using only spatial taxel coordinates. An example of a generic high-level behavior based on two skins with different number of taxels and different taxel distributions are illustrated in § 4.1.

To make practical use of the taxel data, it is commonly necessary to relate this spatial information to the underlying kinematic chain, making it obvious where a taxel is located in relation to the robot's body, e.g., on the upper left arm. The generic robotic skin interface maps individual taxels to points in space relative to elements in the kinematic chain, abstracting away any underlying platform-specific taxel-indexing mechanisms as well as any related connectivity information. The abstract spatial taxel information removes any platform-specific representations and provides a skin representation that can support the implementation of cross-platform capabilities. An example of the forearm skin of the iCub robot represented using the TCI 3-dimensional spatial skin model is presented in **Figure 3**, where each small blue dot is a taxel on the skin. Our work leveraged the existing work of spatial calibration to generate the skin model, in the joint research (Prete et al., 2011; Denei et al., 2015).

In some cases, the three-dimensional position information of each taxel of the robot skin sensors is not available to support the calculations that are needed to present the taxels using the spatial coordinates required by the generic skin interface. In this case, it is time consuming and inaccurate to manually measure the precise 3D positions for all the taxels and instead they have to be approximated.

In our experience, for a taxel t, the displacement along the limb and the transverse angle of the taxel location relative to the limb orientation are typically more important than its distance from the central axis of the limb. As a consequence, we have approximated the spatial distribution of the taxels on the NAO robot skin using truncated cone. Given the estimated dimensions of the truncated cone, our 3D skin model can approximate the real taxel distribution. As a consequence, in order to support a TCI representation of the NAO skin sensors, the position P_t of a taxel t is modeled as a vector $<x_i, \theta, r_t>$ formalized in equation (3).

$$SKIN = \left\langle P_t|_{t=1}^{n} \right\rangle = \left\langle \langle x_t, \theta_t, r_t \rangle|_{t=1}^{n} \right\rangle \tag{3}$$

In equation (3), the value x_t represents the displacement of the taxel along the central axis of the element of the kinematic chain on which it is located. The value θ_t represents the displacement angle within the transverse plane relative to the orientation of the element, along which the taxel is located. The value r_t represents the distance (radius) from the central axis at which the taxel is located. With this taxel model, a generic robot skin sensor can be approximated and represented within the TCI framework. The truncated cone model is presented graphically in **Figure 4**.

Our truncated cone representation has the added benefit that the radius r_t need not be explicitly represented but can be calculated from the displacement x_t. As a consequence, we end up with the approximated skin representation $\overline{SKIN}$ given in equation (4). Admittedly, this model has its limitations on a robot with complex tactile surface where multiple curvatures are present. More investigations are needed as our future work.

$$\overline{SKIN} = \left\langle P_t|_{t=1}^{n} \right\rangle = \left\langle \langle x_t, \theta_t \rangle|_{t=1}^{n} \right\rangle \tag{4}$$

4. A TCI CASE STUDY: HUMANOID TACTILE WITHDRAWAL REFLEXES

In the previous sections we presented the TCI architecture and its two core interfaces, the *Generic Actuator Interface* and the *Generic Robot Skin Interface*. This section presents the application of the TCI framework to the design and the implementation of a cross-platform tactile withdrawal reflex for humanoid robots. The behavior was realized and demonstrated on two physical robots, the iCub and the NAO.

Robotic tactile withdrawal reflexes aim to improve the safety of human–robot interaction by reducing the potential for harm to humans and robots. Safe-path planning, padding, compliant limbs, and withdrawal reflexes all contribute at strategic point in

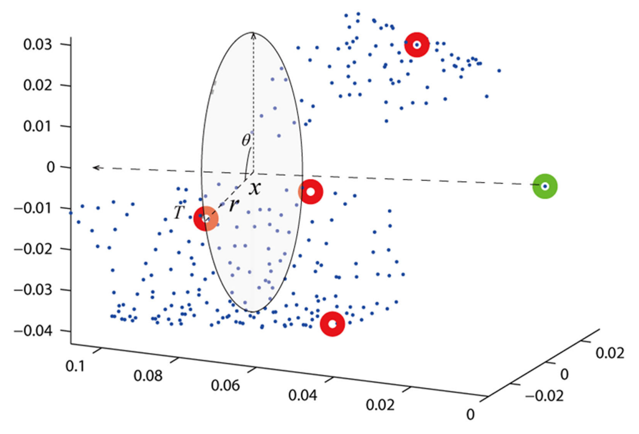

FIGURE 3 | An example of 3-dimensional generic robotic skin model: the forearm skin of iCub.

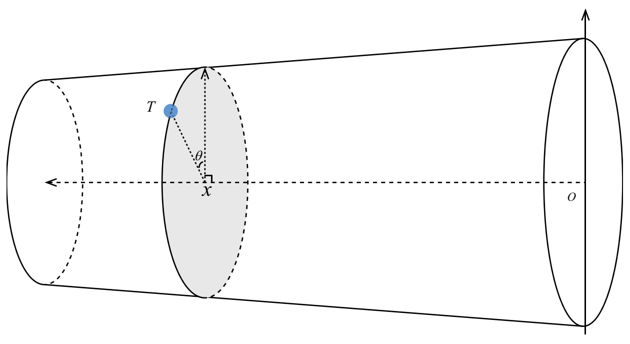

FIGURE 4 | The simplified model of generic robotic skin. The shape of the skin is arbitrarily approximated as a truncated cone.

time to improve safety and increase the scope for the application of human robots in unstructured human environments.

Based on a new robot skin sensor that covers large areas of a robot (Schmitz et al., 2011) and information about the mechanisms supporting human withdrawal reflexes, Dahl and Paraschos (2012) proposed a force–distance reflex model for humanoid robots. This model represents the reflex motion as a base motions moderated by two discount factors: the force of the impact and the distance between the stimulus and the center of the closest *reflex receptive field* (RRF). The RRFs will be discussed in details in §4.1. The base motions for the robot withdrawal reflexes were established through a set of experiments capturing reflex motions from humans using a motion capture suit (Dahl and Palmer, 2010). In our previous research, withdrawal reflex data was obtained using five stimulation locations on the upper and lower arm (four on the lower arm and one on the upper arm).

4.1. The Force–Distance Reflex Model

The force–distance (FD) reflex model is inspired by the concept of reflex receptive fields (RRFs), where each reflex has a trigger in the form of a continuous area on the surface of the skin, within which stimulation will provoke a reflex motion. The strongest response, i.e., the largest motion, is produced when the stimulus is in the center of the field. The strength of the response is gradually reduced as the distance between the stimulation point and the center of the field increases. The edge of the field acts as a threshold, beyond which no motion is triggered. In addition to being sensitive to the location of the stimulus, the size of the response under the FD model is also sensitive to the intensity of the stimulus, i.e., its force. The force–distance model is formalized in equation (5).

$$\theta_{i,j} = \begin{cases} \phi F_i \psi d_i \Theta_{i,j} & \text{if } d_i < r_i \text{ and } F_i > \delta_f \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

The FD reflex model is essentially a set of mappings from tactile stimulus to robotic reflex motions. Each mapping ***i*** defines the radius r_i of a circular receptive field centered at location C_i. The actual reflex motion $\theta_{i,j}$ is a vector of angle displacements, which is a moderation of a vector of base angle displacements $\Theta_{i,j}$. The base angle displacements were obtained by analyzing reflex motions captures from the human subjects. They are discounted by two factors ϕ and ψ, respectively, corresponding to the stimulus force F and the stimulus distance d to the center C_i. Simulation of a location, not covered by a receptive field, will not trigger a response.

Using the tactile capabilities interface, the FD reflex model can be made generic by representing the values involved using values and structures available in the generic robotic skin model discussed in §3.2 and the Generic Actuator Interface presented in § 3.1.

Reproducing the reflex motions $\theta_{i,j}$ using the tactile capabilities interface required us to calculate the stimulation distance d using the generic spatial taxel model. The FD reflex model, rewritten using this representation, is presented in equation (6). § 4.2 discusses, in detail, the implementation of the robotic reflexes under TCI.

$$\theta_{i,j} = \begin{cases} \phi \bar{F}_i \psi \left|\bar{P}_i - C_i\right| \Theta_{i,j} & \text{if } \left|\bar{P}_i - C_i\right| < r_i \text{ and } \bar{F}_i > \delta_f \\ 0 & \text{otherwise} \end{cases}$$

$$\left|\bar{P}_i - C_i\right| = \left|\langle x, r\sin\theta, r\cos\theta\rangle - \langle x_i, r_i\sin\theta_i, r_i\cos\theta_i\rangle\right| \tag{6}$$

In equation (6), $\bar{P}_i$ denotes the position of the center of the pressure and $\bar{F}_i$ is the average force of the triggered taxels C_i of the receptive field and the stimulus point $\bar{P}_i$. In practice, as taxels are close to each other, a stimulation typically triggers multiple taxels simultaneously. For a stimulation with n triggered taxels, F_t and P_t are the force and the position of a taxel t, respectively. $\bar{P}_i$ is the center of the stimulation weighted by the force of the triggered taxels. Similarly, $\bar{F}_i$ is the average taxel force weighted by the distance to $\bar{P}_i$. The position of the center of the pressure $\bar{P}_i$ and the average force $\bar{F}_i$ are formalized in equation (7).

$$\bar{P}_i = \frac{\sum_{t=1}^{n} F_t \cdot P_t}{\sum_{t=1}^{n} F_t}, \bar{F}_i = \frac{\sum_{t=1}^{n} F_t \cdot (2r_i - |P_t - \bar{P}_i|)}{\sum_{t=1}^{n} (2r_i - |P_t - \bar{P}_i|)} \tag{7}$$

In equation (7), $2r_i$ denotes the diameter of the receptive circle, which is the maximum geometrical distance for which a tactile stimulation can produce a response.

4.2. The Implementation of the Robotic Reflexes

This section presents the implementations of the force–distance reflex model on two real robot platforms, NAO and iCub using the tactile capabilities interface. The problem of using the same reflex module on two different robots to establish the same robotic withdrawal reflex motions is discussed in detail. A NAO robot is a 4.5-kg, 58-cm tall humanoid robot designed and manufactured by Aldebaran Robotics. It has 21 DoF (for the RoboCup edition) and is equipped with a range of sensors including two cameras, sonars, touch sensors, and accelerometers. An iCub is a 1-m high humanoid robot test-bed for research on cognitive robot behaviors. The robot is open source both in terms of the hardware design and the software resources. An iCub has 53 DoF and weighs around 22 kg. Both the NAO and iCub robots used for the work presented here were equipped with robot skin sensors developed under our previous joint research (Cannata et al., 2012). The robot reflexes were implemented and evaluated on both the simulated and the real robots. The FD model implementation on the iCub is illustrated in **Figure 5**. The blue dots are taxels of the forearm skin. When stimulations occur within receptive fields, i.e., the four circular red areas, the *Reflex* module produces the generic withdrawal actions. The actions are further translated to iCub specific commands by TCI. According to the FD model (equation (6)), iCub does not response to the stimulations located outside of the receptive fields. Similarly, the FD model on the real NAO is illustrated in **Figure 1B**.

Using the layered structure presented in **Figure 2**, the FD reflex model was implemented as a *Reflex* module located in layer L4. The actual robots APIs in layer L1 send raw tactile data to the YARP interface in layer L2. Within the TCI structures in layer L3, the *Skin Interface* module translates the raw skin data into the abstract spatial representation presented in §3.2.

The high-level *Reflex* module uses the abstracted skin data to calculate the moderated reflex motion $\theta_{i,j}$ using equation (6). Correspondingly, controlling the specific actuators is the reverse process. The reflex motion $\theta_{i,j}$ found by the layer L4 module is, sent down to the generic actuator interface located in layer L3 which again translates the generic motion to platform-specific commands. The YARP layer, L2, further translates the high-level commands to the actual actuators on the specific robot in the bottom layer, L1. **Figure 6** illustrates the robot reflexes produced by the same *Reflex* module being executed on simulated NAO and iCub robots through TCI. Two stimulations are simulated on the bottom and inside parts of the forearms of the two robots, respectively.

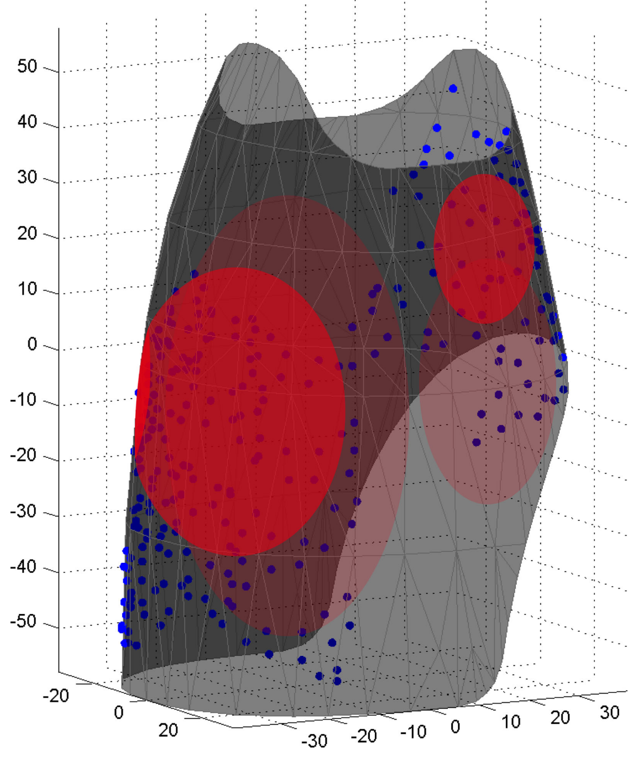

FIGURE 5 | An example of the 3D receptive fields of iCub forearm using generic robotic skin interface. The blue dots are the taxels, and the four circular areas are receptive fields.

5. CONCLUSION AND FUTURE WORK

This article presented a cross-platform tactile capabilities interface (TCI) for development of humanoid tactile capabilities. TCI promotes reuse of high-level modules by providing abstract hardware-independent representations of humanoid robot sensors and actuators. These representations allow control software to focus on the platform-independent elements of the control algorithms, delegating the translation of these abstracted representations to platform-specific commands, to the lower levels of the TCI infrastructure. The literature related to generic interfaces and current approaches to improve robot software reusability was reviewed in §2. As an important method to promote reusability, the state-of-the-art robotic middleware platforms are compared. In §3, a layered architecture for reusable generic robotic modulars was proposed, where TCI is used as an "interpreter" between high-level modules and YARP. TCI contains two core interfaces, a generic actuator interface (§3.1), which solves the configuration differences of the low-level actuators, and a generic robotic skin interface (§3.2) that abstracts the skin data. The article has presented the case study of generic humanoid tactile withdrawal reflexes. The force–distance (FD) reflex model is extended so as to be used under the TCI framework. The FD model has been implemented, and the same module was used to control two different real robots, the NAO and the iCub robots, equipped with different skin sensors. Experiments show that generic reflex motions have been successfully realized under TCI both in simulated environments and on real robots.

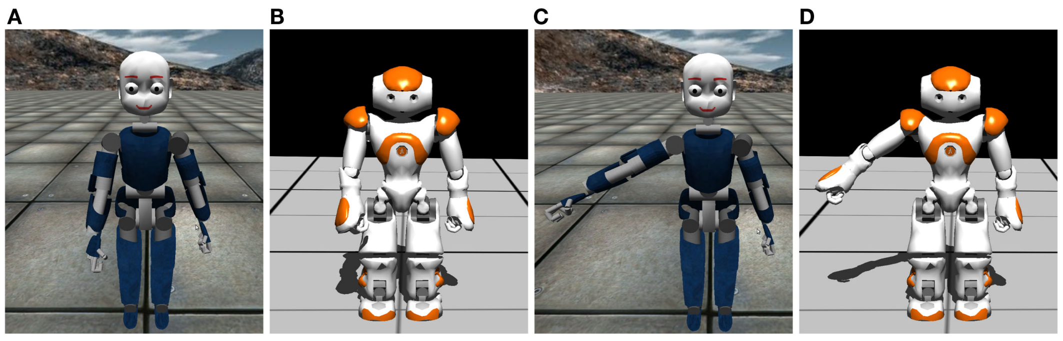

FIGURE 6 | The generic robotic reflexes produced and executed by the same *Reflex* module through TCI. Two stimulations are simulated on the NAO and iCub robots, respectively. Images **(A,B)** illustrate the reflexes from the stimulations located on the bottom of the forearms, while stimulations illustrated in images **(C,D)** are inside of the forearms.

The tactile capabilities interface is our first attempt at developing a cross-platform humanoid robot interface. Currently, only position-based actuator control is implemented and evaluated, although the framework is ready to integrate other types of actuator controls such as velocity control and torque control. Another limitation of our interface is that it does not provide an abstraction for tactile force. This is because in the study case, both the platforms used the same type of the tactile sensors and so the intensity representations are identical. Extra transformations will be needed if the comparison of the forces is required.

Our aim for the future work is to further investigate other control methods and other parts of common humanoid robots in the TCI framework. Also, the integration of forward and inverse kinematics is to be addressed. This feature will remove the requirement for platform-specific kinematics calculations if more sophisticated behaviors are needed. Recently, some research on defecting the direction of tactile force has also been proposed (Fumagalli et al., 2012; Stassi et al., 2014), and this could also become an extension to our current interface.

AUTHOR CONTRIBUTIONS

TD lead the research work presented, contributed to the development of the underlying ideas, wrote parts of the article, and reviewed it before submission.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Commission's Seventh Framework Programme (FP7/2007-2013) under grant agreement ROBOSKIN ICTFP7-231500. The authors would also like to thank Alexandros Paraschos for his contributions to the development of the NaoYarp software.

REFERENCES

Ambrose, R., Wilcox, B., Reed, B., Matthies, L., Lavery, D., and Korsmeyer, D. (2010). *Robotics, Tele-Robotics and Autonomous Systems Roadmap*. Technical Report. Washington, DC: National Aeronautics and Space Administration (NASA).

Anderson, M., and Thomaz, A. L. (2010). Enabling intelligence through middleware: report of the AAAI 2010 workshop. *AI Mag.* 32, 87–90. doi:10.1609/aimag.v32i1.2339

Atkeson, C. G. (1989). Learning arm kinematics and dynamics. *Annu. Rev. Neurosci.* 12, 157–183. doi:10.1146/annurev.ne.12.030189.001105

Bednarz, T., James, C., Caris, C., Haustein, K., Adcock, M., and Gunn, C. (2011). "Applications of networked virtual reality for tele-operation and tele-assistance systems in the mining industry," in *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry, VRCAI '11* (New York, NY: ACM), 459–462.

Brugali, D., and Scandurra, P. (2009). Component-based robotic engineering (part I). *Rob. Autom. Mag. IEEE* 16, 84–96. doi:10.1109/MRA.2009.934837

Bruyninckx, H. (2001). "Open robot control software: the OROCOS project," in *IEEE International Conference on Robotics and Automation (ICRA)*, Vol. 3 (Seoul: IEEE), 2523–2528.

Bruyninckx, H., Soetens, P., and Koninckx, B. (2003). "The real-time motion control core of the OROCOS project," in *IEEE International Conference on Robotics and Automation (ICRA)*, Vol. 2 (Taipei: IEEE), 2766–2771.

Caligiore, D., Guglielmelli, E., Borghi, A., Parisi, D., and Baldassarre, G. (2010). "A reinforcement learning model of reaching integrating kinematic and dynamic control in a simulated arm robot," in *IEEE 9th International Conference on Development and Learning (ICDL)* (Ann Arbor, MI: IEEE), 211–218.

Cannata, G., Mastrogiovanni, F., Metta, G., and Natale, L. (2012). "Advances in tactile sensing and touch based human-robot interaction," in *7th ACM/IEEE International Conference on Human-Robot Interaction (HRI)* (Boston, MA: IEEE), 489–490.

Crick, C., Jay, G., Osentoski, S., and Jenkins, O. C. (2012). "ROS and rosbridge: roboticists out of the loop," in *Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-Robot Interaction, HRI '12* (New York, NY: ACM), 493–494.

Dahiya, R., Metta, G., Valle, M., and Sandini, G. (2010). Tactile sensing – from humans to humanoids. *IEEE Trans. Robot.* 26, 1–20. doi:10.1109/TRO.2009.2033627

Dahl, T., and Paraschos, A. (2012). "A force-distance model of humanoid arm withdrawal reflexes," in *Advances in Autonomous Robotics, Volume 7429 of Lecture Notes in Computer Science*, eds G. Herrmann, M. Studley, M. Pearson, A. Conn, C. Melhuish, M. Witkowski, J.-H. Kim, and P. Vadakkepat (Berlin, Heidelberg: Springer), 13–24.

Dahl, T. S., and Palmer, A. (2010). "Touch-triggered protective reflexes for safer robots," in *International Symposium on New Frontiers in Human-Robot Interaction (AISB'10)* (Leicester: AISB), 27–33.

Denei, S., Mastrogiovanni, F., and Cannata, G. (2015). Towards the creation of tactile maps for robots and their use in robot contact motion control. *Rob. Auton. Syst.* 63, 293–308. doi:10.1016/j.robot.2014.09.011

Diaz-Calderon, A., Nesnas, I. A. D., Nayar, H. D., and Kim, W. S. (2006). Towards a unified representation of mechanisms for robotic control software. *Int. J. Adv. Rob. Syst.* 3, 61–66. doi:10.5772/5757

Fumagalli, M., Ivaldi, S., Randazzo, M., Natale, L., Metta, G., Sandini, G., et al. (2012). Force feedback exploiting tactile and proximal force/torque sensing. *Auton. Robots* 33, 381–398. doi:10.1007/s10514-012-9291-2

Gamez, D., Fidjeland, A. K., and Lazdins, E. (2012). iSpike: a spiking neural interface for the iCub robot. *Bioinspir. Biomim.* 7, 025008. doi:10.1088/1748-3182/7/2/025008

Gerkey, B. P., Vaughan, R. T., and Howard, A. (2003). "The Player/Stage Project: tools for multi-robot and distributed sensor systems," in *The International Conference on Advanced Robotics* (Coimbra: IEEE), 317–323.

Huang, A., Olson, E., and Moore, D. (2010). "LCM: lightweight communications and marshalling," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Taipei: IEEE), 4057–4062.

Huang, Q., Yokoi, K., Kajita, S., Kaneko, K., Arai, H., Koyachi, N., et al. (2001). Planning walking patterns for a biped robot. *IEEE Trans. Rob. Autom.* 17, 280–289. doi:10.1109/70.938385

Ma, J. (2011). *Learning and Cooperation in Multi-agent Systems*. Ph.D. thesis, Department of Computer Science, University of Oxford, Oxford.

Ma, J., and Cameron, S. (2009a). "Learning robust and energy-efficient biped walking patterns using QWalking," in *The Twelfth International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines (CLAWAR 2009)*, eds O. Tosun, H. L. Akin, M. O. Tokhi, and G. S. Virk (Istanbul: World Scientific), 599–606.

Ma, J., and Cameron, S. (2009b). "Learning to walk: a model-free biped locomotion approach," in *Towards Autonomous Robotic Systems (TAROS)*, eds T. Kyriacou, U. Nehmzow, C. Melhuish, and M. Witkowski (Londonderry: University of Ulster), 229–235.

Ma, J., and Cameron, S. (2011). "Learning fast walking patterns for a NAO robot using Qwalking," in *Proceedings of the Fourteenth International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines (CLAWAR2011)* (Paris: UPMC University), 257–266.

Maes, P. (1991). "The agent network architecture," in *Special Interest Group on Artificial Intelligence (SIGART) Bulletin*, Vol. 2 (New York, NY: ACM), 115–120.

Martins, N., Bertol, D., Lombardi, W., Pieri, E., and Dias, M. (2008). "Neural control applied to the problem of trajectory tracking of mobile robots with uncertainties," in *10th Brazilian Symposium on Neural Networks, 2008 (SBRN'08)* (Salvador: IEEE), 117–122.

McMahan, W., Gewirtz, J., Standish, D., Martin, P., Kunkel, J., Lilavois, M., et al. (2011). Tool contact acceleration feedback for telerobotic surgery. *Haptics IEEE Trans.* 4, 210–220. doi:10.1109/TOH.2011.31

Merzouki, R., Fawaz, K., and Ould-Bouamama, B. (2010). Hybrid fault diagnosis for telerobotics system. *Mechatronics* 20, 729–738. doi:10.1016/j.mechatronics.2010.01.009

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Rob. Syst.* 3, 43–48. doi:10.5772/5761

Namoshe, M., Tlale, N., Kumile, C., and Bright, G. (2008). "Open middleware for robotics," in *The International Conference on Mechatronics and Machine Vision in Practice (M2VIP)* (Auckland: IEEE), 189–194.

Nesnas, I. A., Simmons, R., Gaines, D., Kunz, C., Diaz-Calderon, A., Estlin, T., et al. (2006). CLARAty: challenges and steps toward reusable robotic software. *Int. J. Adv. Rob. Syst.* 3, 23–30. doi:10.5772/5766

Prete, A. D., Denei, S., Natale, L., Mastrogiovanni, F., Nori, F., Cannata, G., et al. (2011). "Skin spatial calibration using force/torque measurements," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on* (San Francisco, CA: IEEE), 3694–3700.

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software* (Kobe: IEEE), 5.

Schmitz, A., Maiolino, P., Maggiali, M., Natale, L., Cannata, G., and Metta, G. (2011). Methods and technologies for the implementation of large-scale robot tactile sensors. *IEEE Trans. Robot.* 27, 389–400. doi:10.1109/TRO.2011.2132930

Stassi, S., Cauda, V., Canavese, G., and Pirri, C. F. (2014). Flexible tactile sensing based on piezoresistive composites: a review. *Sensors* 14, 5296–5332. doi:10.3390/s140305296

Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. Cambridge, MA: MIT Press.

Stone, P., and McAllester, D. (2001). "An architecture for action selection in robotic soccer," in *The Fifth International Conference on Autonomous Agents*, eds E. Andre, S. Sen, C. Frasson, and J. P. Müller (New York, NY: ACM), 316–323.

Wooden, D., Malchano, M., Blankespoor, K., Howardy, A., Rizzi, A., and Raibert, M. (2010). "Autonomous navigation for BigDog," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on* (Anchorage: IEEE), 4736–4741.

Yoshida, E., Belousov, I., Esteves, C., and Laumond, J.-P. (2005). "Humanoid motion planning for dynamic tasks," in *The IEEE-RAS International Conference on Humanoid Robots* (Tsukuba: IEEE), 1–6.

Youssefi, S., Denei, S., Mastrogiovanni, F., and Cannata, G. (2014). "Skinware: a real-time middleware for acquisition of tactile data from large scale robotic skins," in *2014 IEEE International Conference on Robotics and Automation (ICRA)* (Hong Kong: IEEE), 6421–6426.

Youssefi, S., Denei, S., Mastrogiovanni, F., and Cannata, G. (2015a). A real-time data acquisition and processing framework for large-scale robot skin. *Rob. Auton. Syst.* 68, 86–103. doi:10.1016/j.robot.2015.01.009

Youssefi, S., Denei, S., Mastrogiovanni, F., and Cannata, G. (2015b). Skinware 2.0: a real-time middleware for robot skin. *SoftwareX* 3, 6–12. doi:10.1016/j.softx.2015.09.001

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.