Proceedings of the ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
IDETC/CIE 2011
August 28-31, 2011, Washington, DC, USA

DETC2011-47185

A PLUG AND PLAY MIDDLEWARE FOR SENSORY MODULES, ACTUATION PLATFORMS AND TASK DESCRIPTIONS IN ROBOTIC MANIPULATION PLATFORMS

Ayssam Elkady
School of Engineering
University of Bridgeport
Bridgeport, Connecticut 06604
Email: ayssam.elkady@gmail.com

Jovin Joy
School of Engineering
University of Bridgeport
Bridgeport, Connecticut 06604
Email: jovinj@gmail.com

Tarek Sobh
School of Engineering
University of Bridgeport
Bridgeport, Connecticut 06604
Email: sobh@bridgeport.edu

ABSTRACT

We are developing a framework (*RISCWare*) for the modular design and integration of sensory modules, actuation platforms, and task descriptions that will be implemented as a tool to reduce efforts in designing and utilizing robotic platforms. The framework is used to customize robotic platforms by simply defining the available sensing devices, actuation platforms, and required tasks. The main purpose for designing this framework is to reduce the time and complexity of the development of robotic software and maintenance costs, and to improve code and component reusability. Usage of the proposed framework prevents the need to redesign or rewrite algorithms or applications due to changes in the robot's platform, operating systems, or the introduction of new functionalities.

In this paper, the *RISCWare* framework is developed and described. *RISCWare* is a robotic middleware used for the integration of heterogeneous robotic components. *RISCWare* consists of three modules. The first module is the sensory module, which represents sensors that collect information about the remote or local environment. The platform module defines the robotic platforms and actuation methods. The last module is the task-description module, which defines the tasks and applications that the platforms will perform such as teleoperation, navigation, obstacle avoidance, manipulation, 3-D reconstruction, and map building.

The plug-and-play approach is one of the key features of *RISCWare*, which allows auto-detection and auto-reconfiguration of the attached standardized components (hardware and software) according to current system configurations. These components can be dynamically available or unavailable. Dynamic reconfiguration provides the facility to modify a system during its execution and can be used to apply patches and updates, to implement adaptive systems, or to support third-party modules. This automatic detection and reconfiguration of devices and driver software makes it easier and more efficient for end users to add and use new devices and software applications. In addition, the software components should be written in a flexible way to get better usage of the hardware resource and also they should be easy to install/uninstall.

Several experiments, performed on the *RISCbot II* mobile manipulation platform, are described and implemented to evaluate the *RISCWare* framework with respect to applicability and resource utilization.

INTRODUCTION

An autonomous robot framework consists of heterogeneous kinds of components such as the actuators that are used to allow movement and convert commands into actions, the sensors which are used to retrieve information from their environments, and the software components that control the actuation and sensors.

Robot middleware is an abstraction layer residing between the operation system and the software applications (as shown in Figure 1). It is designed to manage the heterogeneity of the hard-

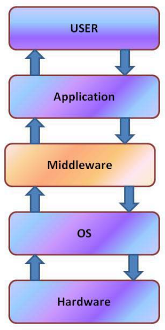

FIGURE 1. Middleware layers.

ware, improve software application quality, simplify software design, and reduce development costs. A developer needs only to build his logic or algorithm as a component, after which his component can be combined and integrated with other existing components. Furthermore, if he wants to modify and improve his component, he needs only to replace the old one with the new one. Therefore, experiment efficiency will improve. In [1], the authors outline some of the problems that face the development of a widely-accepted set of middleware for robotics.

This work addresses the development of a framework (*RISCWare*) for modular design of sensory modules, actuation platforms, and task descriptions that will be implemented as a middleware to reduce and streamline efforts in designing robotic platforms. This framework will be used to customize any robotic platform by simply defining the available different sensing devices, actuation platforms, and required tasks. In addition, this framework will significantly increase the capability of robotic industries in the analysis, design, and development of autonomous mobile platforms.

RISCWare consists of three modules (as shown in Figure 2). The first module encapsulates the sensing devices that gather environmental information, such as infrared sensors, ultrasonic sensors, laser rangefinder and cameras. The framework user should provide the required parameters for all available sensors. The user should also describe the platforms and actuation methods (we have already built a sample platform, the *RISCbot II* mobile manipulator). The last module encapsulates the tasks that the platform will perform such as: teleoperation, navigation, obstacle avoidance, and map building. Our framework will provide some predefined tasks and will allow the users to define new tasks. We have already implemented some tasks including: teleoperation, navigation, manipulation and obstacle avoidance.

RISCWare is implemented as a messaging system because messaging provides a high degree of decoupling between components, so it is used for the integration of heterogeneous system. In addition, messaging offers the ability to process requests asyn-

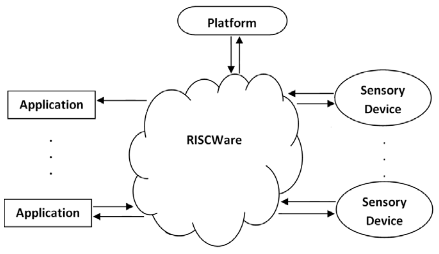

FIGURE 2. Overview of the the *RISCWare* framework.

chronously to increase the performance of the system and reduce system bottlenecks. *RISCWare* ensures that messages are properly distributed among applications. Furthermore, it provides fault tolerance, load balancing, and scalability.

The objective is to design a middleware framework to allow any user, when he gets new sensors, tasks or actuation to just plug it in, hook up a few cables, and the device will work perfectly. Furthermore, the user is able to easily install and uninstall any hardware/software component at any time. Furthermore, when the hardware devices are plugged into the framework, they are automatically detected by the middleware, which loads the appropriate software and makes the device available for applications to use. This automatic detection and configuration of devices make it easy for the end users to add and use the new devices and software applications. In addition, the software components should be written in a flexible way to get the better usage of the hardware resource and also they should be easy to installl/uninstall.

Section 2 introduces prior work, Section 3 presents the features of *RISCWare*, then the architecture of *RISCWare* is presented in Sections 4 and 5. The *RISCWare* components and services are described in Sections 6 and 7. Section 8 provides a description of our own robot *RISCbot II*. Section 9 evaluates *RISCWare* framework with respect to ”A greeting Application”. Finally, Section 10 presents a summary of the work and draws some conclusions.

RELATED WORK

Several robotics middlewares have been developed over recent years. One of the most widely-used is Player [2]. Player is a distributed device repository server for robots, sensors and actuators, divided into several dynamically loadable libraries. Stage, which is the second part of the software, is a graphical, two-dimensional device simulator. It is designed to support research in multi-robot systems through its use of socket-based communication. In addition to Stage, a high-fidelity, three-dimensional simulator, called Gazebo, is available. Robot Operating System (ROS) [3] provides a structured communications layer above the host operating systems of a heterogenous compute cluster, hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. CLARAty [4] was developed to create a reusable robotic framework to reduce the cost of integrating and testing new capabilities and technologies on robotic platforms that are developed at various institutions. It is exclusively targeted to NASA rovers.

There are several other robotics software platforms available such as Miro [5], RT-Middleware (RTM) [6] SmartSoft [7], Orca([8], [9]), OPRoS ([10], [11]), (UPnP) architecture [12], and Microsoft Robotic Studio [13]

A survey of robot development environments (RDEs) by Kramer and Scheutz [14] described nine open source, freely available RDEs for mobile robots. The RDEs were evaluated and compared from various points of view, suggestions were made on how to use the results of the survey. It concluded with a brief discussion of future trends in RDE design. Nader et al. [15] provided a short overview of several research projects in middleware for robotics and their main objective. Nader et al. [16] provided an overview study of networked robot middleware and different criteria for evaluating networked robot middleware.

Finally in [17], some freely available middleware frameworks for robotics were addressed, including their technologies within the field of multi-robot systems.

Features

RISCWare is designed to provide the following features:

1. ***Modularity***: Supports software re-use and abstraction.
2. ***Hardware Architecture Abstraction***: Hides the low-level device-specific details of the device in order to give the developers more convenient, standardized hardware APIs.
3. ***Platform independence and portability***: Users can choose an appropriate platform for running the applications without changing in its internal structure; they need only to change the system's configuration.
4. ***Device independence***: There are multiple vendors for sensors, such as, GPS, sonars, and so forth. The core algorithms of the system should not depend on the specific device.
5. ***Algorithm independence***: It should be possible to develop each algorithm in isolation, that is, in a separate module, and switch the algorithm being used by selecting some configuration values.
6. ***Scalability and upgradeable***: The framework can be rescaled with growing of its components and take full advantage of it. Furthermore, any part of the framework should be able to extend its capabilities without affecting the other parts.
7. ***Ease of use***: The framework is designed to be easy to use.
8. ***Reliable communication for higher priority messages***: *RISCWare* guarantees the delivery of a high priority message, even if partial failure occurs. Guaranteed delivery uses a store-and-forward mechanism, which means that incoming messages will be written out to a persistent store if the intended consumers are not currently available. Persistence increases reliability, but at the expense of performance. Furthermore, Guaranteed Delivery can consume a large amount of disk space scenarios. *RISCWare* allows to configuring of a retry timeout parameter that specifies how long messages are buffered inside the messaging system. *RISCWare* also allows Guaranteed Delivery to be turned off during testing and debugging.
9. ***Flexible architecture***: The framework has no restrictions on the architecture of the control software.
10. ***Real-time system***: Reactiveness of a robot is guaranteed by the real-time system.
11. ***Plug and play***: The components (software and hardware) can be downloaded and installed at or before run-time.
12. ***Security***: Data transportation and user access should be secure so that no one can control the robots other then the user.
13. ***Support for parallelism***: *RISCWare* can perform a number of processes simultaneously and introduce multiple message receivers that can process different messages concurrently.
14. ***Robustness***: Fault detection and recovery capabilities are necessary to provide the framework the ability to be used in real, critical situations. A failure in one module should not damage the whole system. Even if the hardware and software of an autonomous mobile robot are carefully designed, implemented and tested, there is always the possibility of a fault at runtime. It is desirable that the robot be able to detect and localize faults in its system and be able to set the appropriate repair or control actions in order to be able to complete its mission or at least to proceed to a safe mode.
15. ***Distribution system***: The different software modules of an application should be able to exchange data, and be able to run in different machines, from which each one is able to obtain its maximum efficiency.
16. ***Resource sharing***: The sensors and actuators of a robot are treated as shared resources to be used by several software applications.
17. ***Dynamic wiring***: Allows for changing in the configuration of control flow and the data flow at runtime.
18. ***Asynchronous***: The data and control flow are passed asynchronously from one component to others over the middleware. Asynchronous communication means that the sender is not required to wait for the message to be received or handled by the recipient. The sender is free to send the message and continue processing.
19. ***Open source***: the source code of the *RISCWare* will be available for further development.
20. ***Other software engineering aspects***: such as runtime efficiency, reliability, and maintenance.
21. ***Loosely coupled***. In order to make *RISCWare* more loosely coupled, the following dependencies should be removed

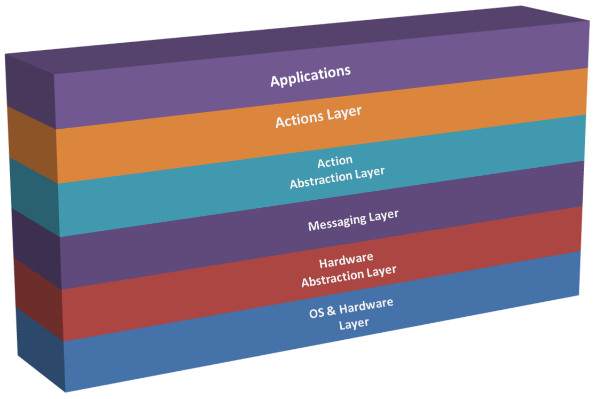

FIGURE 3. The Architecture of *RISCWare* Middleware.

[18]: data format (data representation), location (hard-coded machine addresses) and availability (all components have to be available at the same time). *RISCWare* uses a standard data format that is self-describing and platform independent, such as XML, to remove the data format dependency. To solve the other dependencies, all information is sent to an addressable channel instead of sending information directly to a specific machine. A channel is a logical address that both sender and receiver agree on, without being aware of each other's identity. The channel decouples the sender and the receiver of a message.

RISCWare Layers

The architecture of the middleware has six primary layers for managing the heterogeneity of the hardware and the software, and creating behaviors that will be used by many applications (as shown in Figure 3). Each layer is cleanly separated from the other layers.

OS & Hardware Layer

The *OS & Hardware Layer* (*OSHL*) of robot middleware includes the robot hardware system, composed of a variety of sensors, actuators and other hardware devices, and also the operating system (i.e., Microsoft Windows, UNIX, etc.) that runs on the robot.

Hardware Abstraction Layer

The *Hardware Abstraction Layer* (*HAL*), the platform-dependent part of *RISCWare*, is used to hide the heterogeneity of lower hardware devices and provide a component interface for the upper layers call. *HAL* removes hardware and operating system dependencies between the robot and the application in order to assure portability of the architecture and application programs. It provides access to the sensor data or actuation commands abstracted from the underlying physical connection of the resource. The standard interface to hardware devices takes place through the seven following operations: (All the operations will indicate error conditions if they fail.) Open, Close, Read, Write, Get attributes, Set attributes and Lock.

Messaging Layer

The *Messaging Layer* (*ML*) acts an intermediary to exchange messages between the lower and upper layers. A message consists of two basic parts: the header (which describes the data being transmitted, its origin, its data type, and so on) and the body (data). There are four types of messages, such as the *Command message*, used to invoke a procedure in another application; the *Document message*, used to pass a set of data to another application; the *Event message*, used to notify another application of a change in this application and the *Request-Reply message*, used when an application should send back a reply. The messages are classified into three categories: Simple message (small messages with low delay requirements), realtime message (small message with a certain deadline), and message stream (message sequence with a certain rate). The priority setting of a message can be adjusted an urgent message that should be delivered first. There are 10 levels of priority, ranging from 0 (lowest priority) to 9 (highest priority).

The *Messaging Layer* guarantees the delivery of a high priority message, even if partial failure occurs (as described in the "Features" section).

The *Messaging Layer* supports two types of messaging models: point-to-point (P2P) and publish-and-subscribe (Pub/Sub). The P2P messaging model allows *RISCWare* components to send and receive messages both synchronously and asynchronously via virtual channels known as queues. In P2P, a sender sends messages to a queue that are received by only one receiver (as illustrated in Figure 4). In the Pub/Sub model, a publisher sends messages to a message channel called a topic, after which the messages are received by (one or multiple) subscribers, as illustrated in Figure 5. Messages are automatically broadcast to consumers rather than having request or poll to the topic. The Pub/Sub model is more decoupled than the P2P model in that the publisher is generally unaware of how many subscribers there are or what those subscribers do with the message.

Actions Abstraction Layer

The *Actions Abstraction Layer* (*AAL*) is used to connect a software component to the *ML*, using the channel adapters because the action module and the messaging system are two separate sets of software. Channel adapters hide the details of the *ML* and allow the software application, just knowing that it has a request or piece of data to send to another application, or is expecting them from another application. A channel adapter is a special piece of code customized to both the action module and the mes-

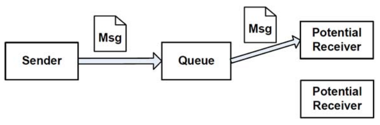

FIGURE 4. Point to Point Model (One to One).

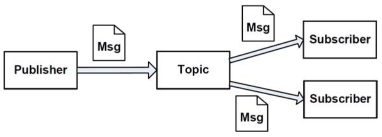

FIGURE 5. Publish and Subscribe Model (One-to-many broadcast).

saging system's client API. A channel adapter is used to send messages or receive them, but one instance does not do both. An adapter is channel-specific, so a single application would use multiple adapters to interface with multiple channels. An application may use more than one adapter to interface to a single channel, usually to support multiple concurrent threads.

Actions Layer

The *Actions Layer* (*AL*) contains the tasks (actions), which are software modules used for sensing, decision-making, and autonomous action such as motion planning, vision, localization, tracking, motion control, etc. *RISCWare* provides a number of often used functional modules such as obstacle avoidance, navigation, logging and visualization facilities modules. A module may communicate with the others using message channels. The Task Manager has been designed in order to allow the user to dynamically load his/her modules, to specify their execution features (i.e. execution period, scheduling policy, priority and so on) and to export the information to be shared among them. The *AL* has three types of execution modes:

1. **Concurrent Execution**: When two or more tasks run simultaneously in parallel, the *TM* fuses the behaviors and removes any conflicts between them.
2. **Time-sharing Execution**: Each task runs exclusively at one time. The *TM* will schedule each task, interrupt running tasks, or resume the suspended tasks on schedule. The *TM* must decide whether or not the current task can be suspended. When the current task is suspended, the *TM* must automatically decide when the robot should resume it. When the current task cannot be suspended, the *TM* rejects the new request or stores this request on a queue for requests awaiting execution.
3. **Sequential Execution**: Each task runs using batch processing.

Each application should provide some additional information about what state is critical for the task. When an application task breaks the critical state requested by the additional information of the counterpart application, the *AL* selects the time-sharing execution. For example, when a robot is required to keep quiet, the robot will not say ”hello” while guiding the user. The other case is interference between both tasks. Interference occurs when both applications access a device simultaneously. For example, a robot cannot shake hands when it holds a cup of coffee in that hand. When both applications access a device at the same time, the *AL* can fuse the motion vectors of each task.

Applications Layer

The application is a set of tasks that work together such as SLAM, Obstacle Avoidance, Navigation, Vision, etc. Every application component should implement its interface. This interface includes the functionalities of starting/stopping a resource, configuring the resource, and connecting a port of the resource to a port of another resource. The Software Assembly Descriptor (SAD) describes a software configuration, the properties and the connections among components. This layer provides the required application program interface (API) to the application layer, such as: installing/uninstalling a robot application, starting/stopping it, registering/unregistering an application, etc.

Architecture of the RISCWare

As shown in Figure 6, the lowest layer of robot middleware is the OSHL. The hardware drivers, used in the second layer (HAL), are designed to hide the low details of the OS and hardware, and publish a message to an appropriate message channel in the ML. In the third layer (ML), message channels provide a very basic form of routing capabilities. A message translator is used to translate the sensor-specific (private) messages to canonical (public) messages. For example, a message translator might transform messages generated by a SONAR1 sonar-type message to a generic message of sonar, and transform from sonar, infrared sensors, or laser rangefinder types message to a generic Proximity message type. A Proximity class defines interface functionality common to sensors like infrareds, sonars, and laser range finders. For the purpose of consistency, the interface through which components can be accessed and controlled is standardized. In the (AAL), channel adapters are used to send messages or receive them to or from the actions, but one instance does not do both.The actions in the (*AL*) consist of some tasks. Each action is implemented as a software module to perform a number of specific tasks used for sensing, decision-making, and autonomous action. The actions can communicate together using message channels. The last layer is the application layer, which consists of some applications such as SLAM, Obstacle avoidance, Navigation, vision, etc. Every application contains some actions working together to perform a certain task. In Figure 6, a robot application ”greeting a person” is composed of Face Detection, Face Recognition, and Text to Speech Modules.

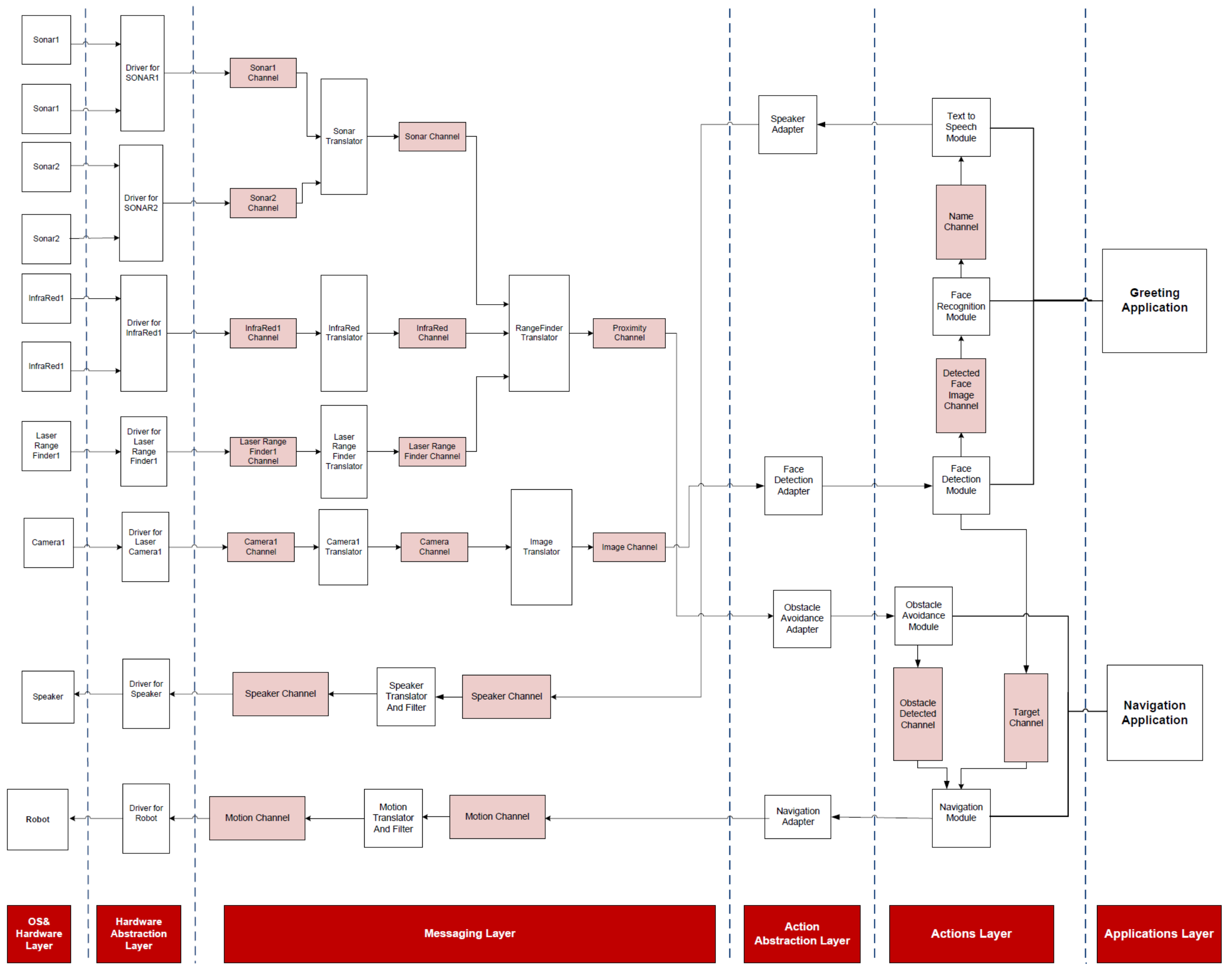

FIGURE 6. An overview of the *RISCWare* middleware

RISCWare Components

Introduction

A *RISCWare*-component is the basic functional module of the *RISCWare* framework, such as: a motor module, a sensor module, and an image processing module. Each component has an XML configuration file to customize the behavior of components, fulfilling the requirement for the reuse of components that third parties be able to compose and customize without ever having to look at the source code. Each component has self-describing capability. Each component should - upon system initialization - announce its presence, register its services, etc. in a coherent way. Furthermore, the component has a PnP capability, which requires that components can be added and removed during system operation, without system reboot. Each component is designed to be dynamically reconfigurable during robot operation. There are four levels of component granularity:

1. ***Atomic component***: An atomic component is a particular type of hardware abstraction that provides the unified interface for upper component calls, such as a motor control atomic component, various sensors control atomic components, etc.
2. ***Composite component***: Several atomic components can be combined into composite components, which provide superior components. For example, a robot chassis consists of several motors and sensors to provide specific service interfaces, such as chassis velocity settings. Through this abstraction, whether the chassis is a four-wheel drive or a two-wheel drive, the superior components can control the robot chassis through the unified service interface.
3. ***Action component***: An action component is a platform-independent robot algorithm or generic robot control algorithm, for example, an obstacle avoidance algorithm and Kalman filter, image processing algorithm, etc.
4. ***Application component***: Application components are the combination of action components needed to complete the

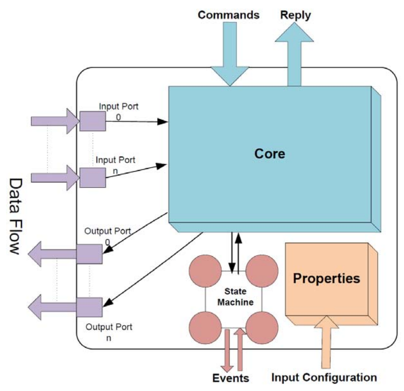

FIGURE 7. The Architecture of the *RISCWare*-component.

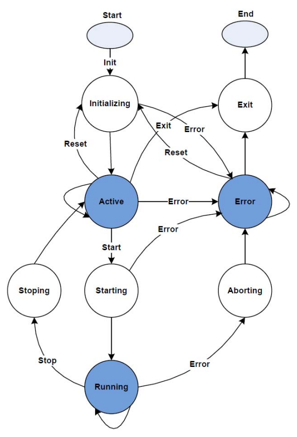

FIGURE 8. The FSM transition of the *RISCWare*-component.

application functions of the robot system.

The architecture of the *RISCWare*-component is shown in Figure 7. The *RISCWare*-component consists of the following objects:

1. ***Core***: The main process unit. It executes the commands, events and software script in the component.
2. ***Input/Odutput data stream ports***: Based on the publisher/subscriber model, the data port is used for exchanging data. An output port (publisher) sends data to all registered subscribers (input ports) through the message channels.
3. ***Properties***: The module's properties can be configurable at runtime.
4. ***Finite State Machine (FSM)***: As shown in figure 8, the *RISCWare*-component has eight states: Initializing, Active, Error, Starting, Running, Aborting, Stoping and Exit. Figure 8 shows the state transition diagram of *RISCWare*'s FSM.
5. ***Commands/Reply port***: This port is used to receive the "Command Messages" from the *RISCWare*. When the component receive a command, it should be processed immediately (not like the data in the data port).
6. ***Event/Reply port***: This port is used to receive the "Event Messages" to update the FSM of the component by changing the current state to the next state of the FSM, based on the received Event. Furthermore, once an event is detected, the automatic notification service will alert the interested component that the event has occurred.

Hardware Driver

The Drivers are divided into two classes: sensor drivers, which read input data from sensors, such as the Sonar and GPS; and control drivers, which control devices, such as left and right wheels, and the robotic arm. Besides hiding the details of communicating with the device, a driver also transforms the data to match units and conventions used by the rest of the system. Hardware drivers can send a message by creating the message in the appropriate format (based on the definition of the message format). For example, a message might contain information on sender, receiver, time stamp and the sensory data then place the message into the communications system. The hardware drivers can receive a message by doing the following: receive the message from the communications system, and then parse the message into its control information and data.

Channels and Filter

As described in [18], message channels provide routing capabilities, similar to pipe symbol in unix. Once a components subscribes to a message channel, it will, by default, consume all messages from that channel. The channels used in *RISCWare*, are called Datatype (the data on a channel has to be of the same type). This is the main reason why *RISCWare* needs many channels. A channel is implemented to be a fixed-sized message buffer, so the old messages are overwritten if the buffer is full. As described in [18], the following issues should be considered in the channel management : response time, message format, message size, message priority, channel volume, channel timeouts, security and channel persistence to store messages in a persistent form.

The filters are connected by channels. Each filter provides a very simple interface; "it receives messages on the inbound pipe, processes the message, and publishes the results to the outbound pipe" [18], as illustrated in Figure 9.

In order to improve the system throughput, parallel processing, as shown in Figure 11, and pipeline can be used, as shown in Figure 10, to allow multiple messages to be processed concurrently. However, this configuration can cause messages to be processed out of order.

Message Translator

As described in [18], a message translator is a special filter used to translate application-specific messages to canonical mes-

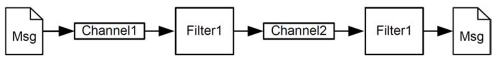

FIGURE 9. Channels combined with filters.

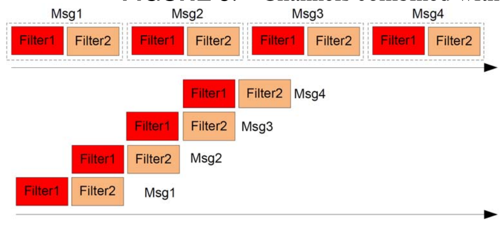

FIGURE 10. Channels processing with pipeline processing.

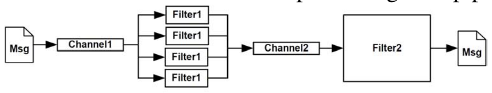

FIGURE 11. Channels processing with parallel processing.

sages which are independent from any specific application to provide all applications the ability to communicate together in common format. "If the internal format of an application changes, only the message translator between the affected application and the coming message channel needs to change while all the other components remain unaffected." [18]

RISCWare Services

Services are shared functions aiding the architects and developers to implement a solution. They are not a part of the core. A service should be a well-defined function and also should be always available to respond to the requests. To be able to use the service, first the new service should add itself in the available services directory of the middleware. Then, each service should declare its interface to allow any application to communicate and use it. The following are the services provided by *RISCWare*:

1. ***Directory***. It is a lookup table used by the *system manager* to store data associated with each component such as the physical and logical address, to track all the components and key information about the system. It is used to automate the action of locating any component.
2. ***Namespace***. Provides a separate namespace for each robot to give the robots the ability to address each other. A messaging bridge is used to connect between robots agents which run *RISCWare* system by replicating messages between systems. A naming service maps names to addresses. A name always exists relative to a naming context. A naming context is itself an object that can be assigned a name. The namespace used in *RISCWare* is modeled as a hierarchical namespace, presented as a directed graph, as shown in Figure 12. The Namespace service is used to support multirobot control.

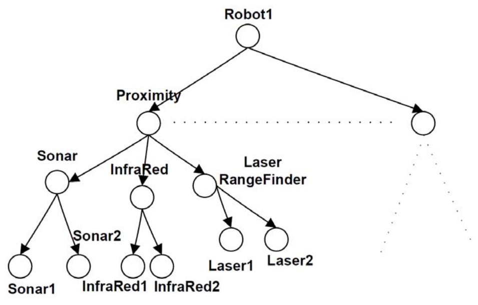

FIGURE 12. A hierarchical namespace used in *RISCWare*.

3. ***Hardware Driver Writing*** : A tool is used to provide facilities for auto-generating code by using a Driver Template provided by the *RISCWare* and the XML Driver file. It is provided by the user, which is a configuration file that contains the required information about the new hardware device.
4. ***System Manager***: Monitors the flow of data, manages the flow of messages through the system, makes sure that all applications and components are available, tracks quality of service (e.g. response times) of an external service, and reports error conditions. There is a Quality of Service (QoS) attached with each component. If the response time exceeds a specified time, it will report that to the *system manager*. Furthermore, the *resource manager* periodically sends a request/reply message to every component to check the availably of this component and whether a failure has occurred or not.
5. ***Security***: *RISCWare* offers authentication, authorization, and secure communications.
6. ***Error Reporting***: The ability to identify and track the error events when a particular problem occurs.

RISCbot II

The *RISCbot II* mobile manipulator has been designed to support our research in algorithms and control for an autonomous mobile manipulator. The objective is to build a hardware platform with redundant kinematic degrees of freedom, a comprehensive sensor suite, and significant end-effector capabilities for manipulation. The *RISCbot II* platform differs from any related robotic platforms as its mobile platform is a wheelchair base. Thus, the *RISCbot II* has the advantages of a wheelchair, such as: high payload, a high speed motor package, Active-Trac and rear caster suspension for outstanding outdoor performance, and adjustable front anti-tips to meet terrain challenges. We used different types of sensors so that *RISCbot II* can perceive its environment with accuracy. Our robot hosts an array of 13 sonars, and 11 infrared proximity sensors above the sonar ring, Hokuyo Scanning Laser Rangefinder, and a wireless network camera.

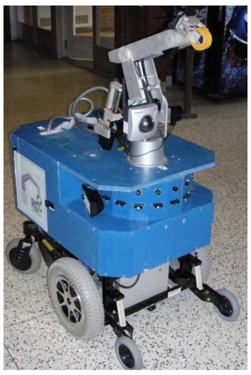

FIGURE 13. A prototype of the RISCbot .

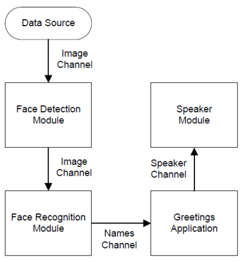

FIGURE 14. Face Recognition and Greeting module integration.

Experiment

A robot application ”greeting a person” is implemented to evaluate the *RISCWare* framework, with respect to applicability. As shown in Figure 6, the ”greeting a person” application is composed of Face Detection, Face Recognition and Text-to-Speech Modules. The Face Detection Module processes images. When it detects the face of a human, the robot approaches the person, detecting collisions with obstacles, and sends the image to the Face Recognition Module to try to recognize the person. If this person is recognized, his name will be sent to the Text to Speech Modules. Each module is developed independently and interacts with each other via *RISCware*, as shown in Figure 14. Both the messages from the Face Detection and Obstacle Avoidance module are fused by the navigation module to control the movement of the robot. Generic Video for Linux (V4L2) is used for video capture. A Logitech Webcam is used to capture images at a rate of 30 fps. Raw image data is transformed into the OpenCV Image format and sent to the Face Detection Module using message channels. The algorithms used for Face Detection and Recognition modules are described in detail in [19].

Results

Face Detection and Recognition modules must be tested simultaneously with the input of the Detection Program fed into the Recognition program. The Face Detection program on an

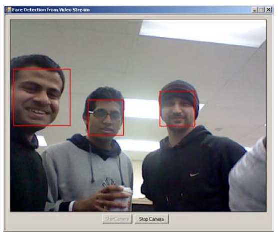

FIGURE 15. Face Detection Module detecting multiple people in a single frame

average detects up to five faces in real time (30 frames/second), running on a Dual Core Intel Processor, therefore bringing the total to 150 images/second. Figure 15 shows a sample output from the Detection Program. The main concerns regarding the number of images that can be detected per frame are the computational requirement and the need to maintain real time performance.

The Recognition Module, on the other hand, can take each of the detected faces and search the database to find possible matches. The initialization process of the Face Recognition database is found to be a processor hog, hence plans to recompute the database values at run time had to be abandoned. Another bottleneck is the total memory requirements for the database, which increases due to storing the feature vectors in uncompressed formats in system memory.

Conclusions

In this paper, the *RISCWare* framework is proposed as a robotic middleware for the modular design of sensory modules, actuation platforms, and task descriptions. This framework will be used to customize robotic platforms by simply defining the available sensing devices, actuation platforms and required tasks. In addition, this framework will significantly increase the capability of robotic industries in the analysis, design, and development of autonomous mobile platforms. *RISCWare* is comprised of three modules. The first module encapsulates the sensors, that gather information about the remote or local environment. The second module defines the platforms, manipulators and actuation methods. The last module describes the tasks that the robotic platforms will perform, such as: teleoperation, navigation, obstacle avoidance, manipulation, 3-D reconstruction and map building. The objective is to design a middleware framework to allow a user to plug in new sensors, tasks or actuation hardware, resulting in a fully functional operational system. Furthermore, the user is able to install and uninstall hardware/software components through the system lifetime with ease and modularity. In addition, when the hardware devices are plugged into the framework, they are automatically detected by the middleware layer, which loads the appropriate software and avails the device for applications usage. This automatic detection and configuration of devices make it efficient and seamless for the end users to add and use the new devices and software applications. Finally, some experiments, performed on the *RISCbot II* mobile manipulator, are described to evaluate the *RISCWare* middleware.

REFERENCES

[1] Smart, W. D., 2007. "Is a common middleware for robotics possible?". In IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07) Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware, E. Prassler, K. Nilsson, and A. Shakhimardanov, eds.

[2] Collett, T. H., MacDonald, B. A., and Gerkey, B. P., 2005. "Player 2.0: Toward a practical robot programming framework". In Proc. of the Australasian Conf. on Robotics and Automation (ACRA).

[3] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y., 2009. "Ros: an open-source robot operating system". In ICRA Workshop on Open Source Software.

[4] Nesnas, I., 2007. "The claraty project: Coping with hardware and software heterogeneity". In *Software Engineering for Experimental Robotics*, D. Brugali, ed., Vol. 30 of *Springer Tracts in Advanced Robotics*. Springer Berlin Heidelberg, Berlin, Heidelberg, ch. 3, pp. 31–70.

[5] Utz, H., Sablatnog, S., Enderle, S., and Kraetzschmar, G., 2002. "Miro - middleware for mobile robot applications". *Robotics and Automation, IEEE Transactions on,* ***18***(4), aug, pp. 493 – 497.

[6] Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., and Yoon, W.-K., 2005. "Rt-middleware: distributed component middleware for rt (robot technology)". In Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on, pp. 3933 – 3938.

[7] Schlegel, C., Hassler, T., Lotz, A., and Steck, A., 2009. "Robotic software systems: From code-driven to model-driven designs". In Advanced Robotics, 2009. ICAR 2009. International Conference on, pp. 1 –8.

[8] Alexei Makarenko, A. B., and Kaupp, T., 2007. "On the benefits of making robotic software frameworks thin". In POn the Benefits of Making Robotic Software Frameworks Thin IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07), San Diego CA, USA, 29 Oct. - 02 Nov. 2007.

[9] Alex Brooks, Tobias Kaupp, A. M. S. W. A. O., 2005. "Towards component-based robotics". In Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05), Edmonton, Canada, 02 Aug. - 06 Aug. 2005.

[10] Jang, C., Song, B., Jung, S., Kim, S., Choi, B., Lee, H.-Y., and Lee, C.-H., 2009. "A development of software component framework for robotic services". *Convergence Information Technology, International Conference on,* ***0***, pp. 1–6.

[11] Jang, C., Lee, S.-I., Jung, S.-W., Song, B., Kim, R., Kim, S., and Lee, C.-H., 2010. "Opros: A new component-based robot software platform". *ETRI Journal,* ***32***, Oct., pp. 646–656.

[12] Ahn, S. C., Lee, J.-W., Lim, K.-W., Ko, H., Kwon, Y.-M., and Kim, H.-G., 2006. "Upnp sdk for robot development". In SICE-ICASE, 2006. International Joint Conference, pp. 363 –368.

[13] Jackson, J., 2007. "Microsoft robotics studio: A technical introduction". *Robotics Automation Magazine, IEEE,* ***14***(4), pp. 82 –87.

[14] Kramer, J., and Scheutz, M., 2007. "Development environments for autonomous mobile robots: A survey". *Auton. Robots,* ***22***(2), pp. 101–132.

[15] Mohamed, N., Al-Jaroodi, J., and Jawhar, I., 2008. "Middleware for robotics: A survey". pp. 736 –742.

[16] Mohamed, N., Al-Jaroodi, J., and Jawhar, I., 2009. "A review of middleware for networked robots". Vol. 9.

[17] Namoshe, M., Tlale, N., Kumile, C., and Bright, G., 2008. "Open middleware for robotics". pp. 189 –194.

[18] Hohpe, G., and Woolf, B., 2003. *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, October.

[19] Elkady, A., Babariya, V., Joy, J., and Sobh, T., 2010. "Modular design and implementation for a sensory-driven mobile manipulation framework". *Journal of Intelligent & Robotic Systems*, pp. 1–27. 10.1007/s10846-010-9454-3.