

Flexible Signal-Oriented Hardware Abstraction for Rapid Prototyping of Robotic Systems

Stefan Jörg, Mathias Nickl, and Gerd Hirzinger
 German Aerospace Center (DLR e.V.)
 Institute of Robotics and Mechatronics
 D - 82234 Wessling, Germany
 stefan.joerg@dlr.de

Abstract—Diffuse and changing specifications for the design of light-weight robots result in high design costs for the desired robotic system, especially the electronic modules and related software drivers. To reduce those costs, we created a flexible robot platform, consisting of FPGA joint modules that are connected by a high speed communication. To fully exploit the hardware flexibility, we introduce a flexible signal-oriented hardware abstraction that is based on a *Signal Flow Oriented Middleware (SFMiddleware)*. *SFMiddleware* enables the transparent integration of changing joint hardware functionality with robot control applications. Utilizing a static system specification approach, we benefit from the abstraction of a middleware without the typical overhead of common middleware implementations. Thus, we achieve a small run-time footprint and control cycles of more than 10 kHz.

I. INTRODUCTION

Our ambition to create advanced light-weight robots for various applications results in diffuse and changing specifications. Hence, we are faced with various and diffuse requirements for the desired robotic system, which causes high design costs. This is especially true for the electronic modules and low-level software drivers. To reduce those design costs, we created a rapid prototyping platform for light-weight robotic systems.

The result is a complex mechatronic system with the following attributes:

- *Distributed*, consists of interacting components which are located throughout the robot
- *Heterogeneous*, due to various computing platforms, co-existing hardware and software implementations, and various implementation methods and languages
- *hard real-time constraints* must be met, we desire control cycles of up to 20 kHz

To handle this complexity, we introduce a *Signal-Oriented Hardware Abstraction Layer (HAL)*. The aim is to integrate seamlessly the distributed hardware components with robot control applications or control modeling tools, such as Real-Time Workshop (see Fig. 2). On the one hand this hardware abstraction must be implemented efficiently to meet the hard real-time constraints, on the other hand it must be as flexible as the hardware represented, i.e. it should be easy to create valid abstractions.

We use the ideas of middleware for the transparent integration of distributed hardware components. For the signal-

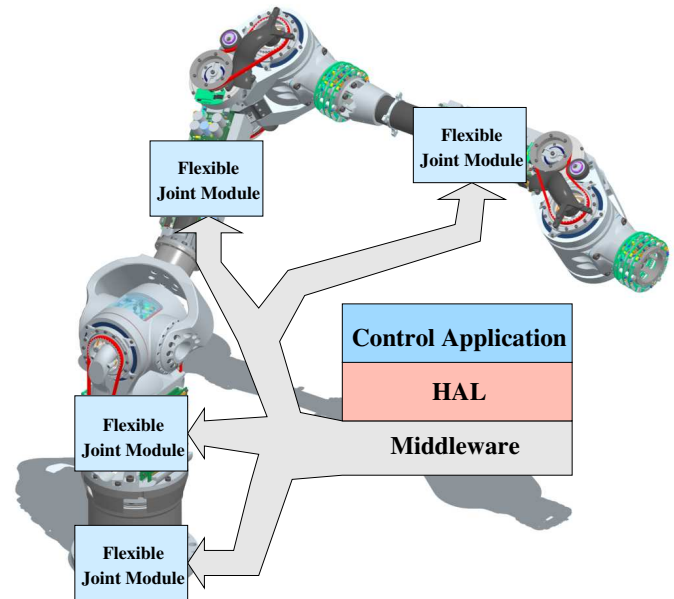


Fig. 1. A Flexible Robot Platform

oriented hardware abstraction we created a Signal Flow Oriented Middleware (SFMiddleware).

SFMiddleware follows a static system specification approach to get an efficient implementation: The entire topology is known at compile time. The resulting small run-time footprint achieves hard real-time constraints and guarantees the economic usage of sparse hardware resources. Thus, we avoid the typical overhead common to middleware implementations, yet we benefit from the abstraction of a middleware:

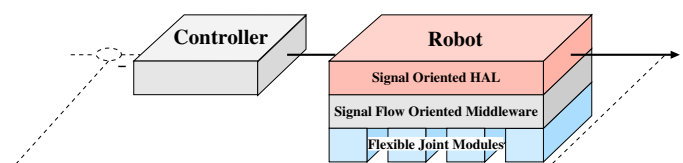


Fig. 2. Hardware Abstraction Built On SFMiddleware

- The formal specification of component interfaces and the transparency of component communication enables concurrent and independent component development
- Component communication can be generated and verified automatically
- Developers concentrate on functionality not infrastructure

In contrast to other real-time middleware, such as RT-CORBA [1] and TAO [2], the scope of *SFMiddleware* is device driver development for the integration of heterogeneous hardware with control applications. Hence, the static system specification is hardly a limitation. Existing approaches for the design of robot controller applications like [3] [4] [5] [6] use robot hardware devices for their applications. Our goal is to build and provide those robot hardware devices and the necessary software drivers.

The following section introduces the flexible platform and explains what is meant by flexible signal-oriented hardware abstraction. Sec. III investigates the impact of distribution on actor-oriented models of control applications. Sec. IV describes the role of *SFMiddleware* in this context. Sec. V describes the concept of a *Flexible Robot Platform*. Sec. VI illustrates our concept of a signal-oriented hardware abstraction with the implementation of a 7DoF robot for medical applications.

II. FLEXIBLE PLATFORM AND SIGNAL-ORIENTED HARDWARE ABSTRACTION

There are many definitions of "platform". These depend on the domain of application [7]. In the field of robotics, a platform is a complex signal-oriented system with heterogeneous hardware modules. Through high integration that modules have become rich in functionality and are configured to a specific application. The interface to such a module should present only the application specific part and not the whole functionality of the module. Ultimately, as for FPGAs, a meaningful interface is only created through configuration. We call a configurable hardware module with a variable interface *Flexible Hardware Module* and a composition thereof *Flexible Platform*.

There are two orthogonal aspects of complexity of a *Flexible Platform*: flexibility and distribution. To handle this complexity, the *Hardware Abstraction Layer* provides an application specific hardware interface of the entire platform, which formulates the view of the application designer on the hardware. So, the HAL combines the distributed functionality and hides the complexity of changing hardware configuration, i.e. hardware abstraction should be as flexible as the hardware platform itself.

The key to a flexible hardware abstraction is to minimize the effort required for its adaptation to hardware changes. To achieve this, those changes must be kept as local as possible and a formal application model should be applied to identify implementation rules.

Signal-oriented modeling tools, like Simulink, have been established to model robot behavior. So, signal-oriented models are suitable application models for robot control and a

proper interface to hardware is also signal-oriented. The design method that is based on functional components connected by signals is called actor-oriented design [8]. In this context, actors are concurrent system components connected by signals, where signals are communication channels which transmit atomic data events called firings [9].

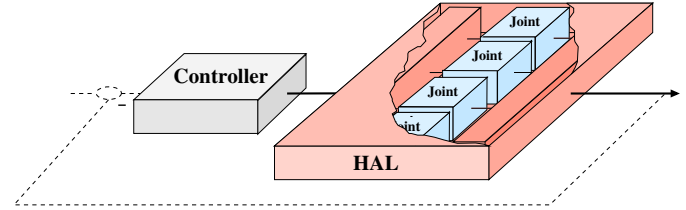


Fig. 3. HAL As Actor Model

With actor-oriented design, signal-oriented hardware abstraction is modeled as an actor representing the hardware functionality. In the context of robot design, the HAL is a hierarchical actor that combines the joints' functionality modeled as actors (see Fig. 3). An implementation of the HAL has to address the problems that arise from the implementation of a distributed actor model:

- Functionality is distributed to heterogeneous platforms
- signals between distributed actors result in real physical communication
- the execution of the separate models must be synchronized

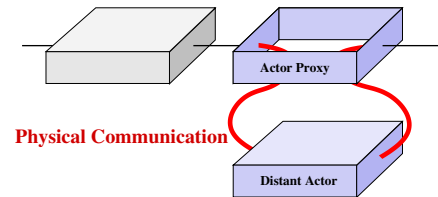


Fig. 4. Integrate Distant Actor Via Proxy

To solve these problems, we apply the ideas of middleware. A proxy interface transparently routes the signals via a communication channel to the distant actor (see Fig. 4). An efficient implementation of this concept is our Signal Flow Oriented Middleware (*SFMiddleware*). It uses CORBA's IDL for formal actor interface specification and a Real-Time Signal Broker (RSB) for the efficient implementation of actor signals.

For rapid prototyping a huge variety of *Flexible Hardware Modules* is available. The concept of an adaptable HAL with a middleware for hardware integration enables the seamless integration of components connected with heterogeneous physical communications in actor-oriented applications. This applies to both commercial components (e.g. motor controllers, sensors) and customized hardware modules (e.g. Flexible Joint Modules, see Sec. V).

III. MAP ACTORS TO DISTRIBUTED COMPUTING PLATFORMS

Actors are concurrent system components connected by signals via actor input and output ports, where signals are considered as abstract non-blocking write and blocking read FIFO communication channels, which transmit atomic data events called firings. An actor awaits a dedicated pattern of firings at the input ports, which synchronizes the output firings, i.e. the entire synchronization of concurrent actors is done by firings [9]. The semantics of actor firing is determined by the *model of computation* and is orthogonal to the *actor composition* and *actor definition*. The *actor composition* defines the hierarchical structure and the signal routing of the actor model, i.e. a hierarchical netlist. The *actor definition* determines the functional behavior of the actor model. For the parameterization of the actor definition, actor interfaces provide configurable attributes.

The selection of a dedicated *model of computation* depends on the nature of the application. Synchronous Data Flow (SDF) is suitable to model the behavior of fixed step control applications and implies that the number of firings transmitted by a signal per time step is statically determined at compile time. Thus, SDF enables an optimized implementation with static scheduling [10].

The implementation of an actor model on distributed computing platforms implies the following difficulties:

- *Distributed actor definition*: Actors are atomic functional units. There is no coupling beside the interaction with signals. Thus, any actor can be mapped to a particular computing platform (FPGA, CPU, DSP), as long as these actor implementations are connected by valid signals. A variety of tools (e.g. Matlab RTW, Xilinx SystemGenerator) are available to compile actor models to dedicated target technologies.
- *Distributed actor composition*: The communication infrastructure of actor-oriented systems has to implement abstract signals on real communication mechanisms (TCP/IP, UDP, SpaceWire, shared memory). Those channel implementations has to fulfill the requirements of actor models, i.e. connecting actors by transmitting firings with negligible latency.
- *Distributed model of computation*: Concurrently running actors are synchronized inherently by patterns of actor firings. Communication errors can cause a violation of real-time constraints. Thus, an appropriate flow control mechanism is needed to observe the real-time behavior and for the synchronization of the actor model execution, e.g. the firings have to be augmented by an appropriate timestamp.
- *Parameterization of distant actors*: If distant actors provide to turn attributes during runtime, a remote access of those attributes is required.

Thus, the communication connecting distributed running partitions of an actor-oriented design can be reduced to the transportation of firings and the remote access to attributes.

IV. SFMIDDLEWARE - INTEGRATE DISTRIBUTED ACTORS

The role of SFMiddleware is the transparent integration of the distributed functionality of heterogeneous hardware modules with a local hardware abstraction. The hardware modules are considered as servers; the application that uses those hardware modules as client. In contrast to other middleware implementations servers are not objects, but actors, and the interfaces connecting client and servers are actor interfaces, not object interfaces (see Fig. 5).

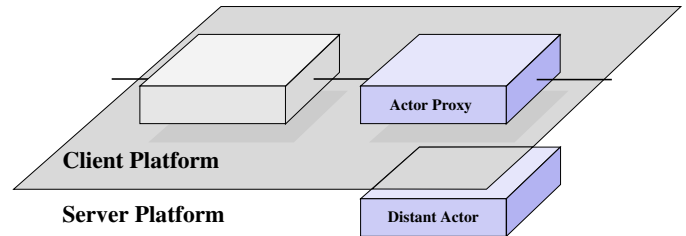


Fig. 5. Distant Actor As Server

The client-server communication has to provide the transport of actor firings and the access to remote actor attributes. An actor firing is an event that has to be transmitted with minimum latency. Diametrical to firings the access to attributes is time-less and needs a maximum of reliability.

Actor interfaces consist of only three basic communication elements: signal sinks, signal sources, and parameters. For the formal description of actor interfaces, SFMiddleware uses CORBA's Interface Definition Language (IDL) [11]. The IDL can be used without modification. Actors are defined as *component*. Actor firings are mapped to CORBA events. The composition type *eventtype* is used for the declaration of input and output firings. The keyword *consumes* defines an event sink and *publishes* an event source. Actor parameters are declared as *attribute* (see Fig. 6).

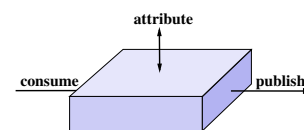


Fig. 6. Basic Communication Elements

The IDL is mapped to the native languages (e.g. C, C++, VHDL) used by the client and server implementations. This involves the mapping of structural elements, such as interface, components, and modules to native language constructs. The IDL types are mapped to native types. Analogous to CORBA, the artefacts of this mapping are called Stubs and Skeletons (see Fig. 7). The Stub is the proxy interface a client uses to transparently access the remote server implementation. The Skeleton binds a server implementation to the communication. Skeletons and Stubs represent native actor interfaces and implement the data marshalling of event and attribute data.

Similar to CORBA's Object Request Broker (ORB) which transparently routes client requests to corresponding servers,

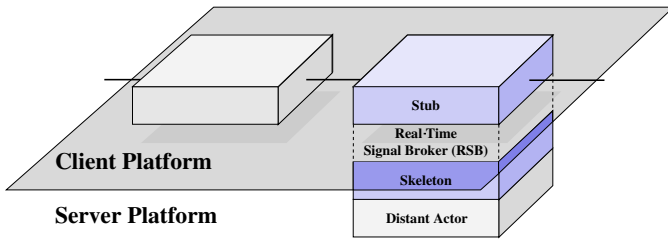


Fig. 7. How SFMiddleware Integrates Distant Actors

SFMiddleware defines a Real-Time Signal Broker (RSB) which efficiently routes actor firings and attribute access requests (see Fig. 7). The basic communication elements of actor communication can be implemented by only four different transactions: consume, publish, set, and get. SFMiddleware's RSB provides these four generic transactions with its *Generic Transaction Layer* (see Fig. 8). This is SFMiddleware's abstraction of communication.

To integrate a physical communication, the mapping of the generic transactions to that communication has to be implemented. The objective is to find an optimized implementation for this mapping, e.g. transactions via shared memory should result in memory access operations only. Once the physical mapping is available, the communication can be used in any SFMiddleware application. Thus, a growing library of different communication platforms can easily be created. One can think of the RSB as the aggregation of all client-server connections in a system. Every connection may use a different physical communication and the RSB provides an abstract communication interface to the client and server implementations.

In contrast to the time-less attribute, the mapping of event sink and event source is dependant on the model of computation: The native event type is composed of the type of the actor firing and the information necessary for the implementation of a distributed model of computation, e.g. a time-stamp. Since the entire synchronization of a distributed actor model can be realized by firings, there is no further

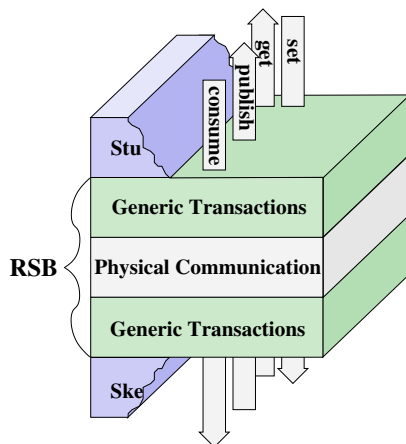


Fig. 8. The Real-Time Signal Broker (RSB)

dependency of communication and model of computation, i.e. the communication refinement is orthogonal to actor definition and model of computation.

The applied application model of signal-oriented systems enables a simple abstraction of communication and an optimized middleware implementation. In contrast to other middleware SFMiddleware is geared towards the integration of hardware modules with strong real-time constraints. Therefore, a static system specification is applied to pay for the costs of abstraction during compile time and thus achieve a small runtime footprint.

V. THE FLEXIBLE ROBOT PLATFORM

The *Flexible Robot Platform* is a *Flexible Platform* for the prototyping of novel robots to enable highly integrated systems while confronted with short development cycles. The idea is, to shorten development time by reusing a stable hardware platform, which enables the effortless adaptation to new specifications, e.g. the integration of new sensors. Thus, the application specific elements have to be changed for the development of a new robot.

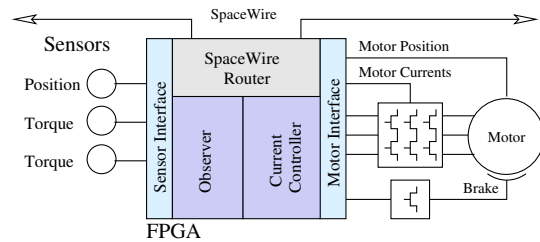


Fig. 9. Light-Weight Robot Joint Module

The *Flexible Robot Platform* is a network of *Flexible Joint Modules* connected by a packet oriented and collision free communication. High communication bandwidth and negligible latency enable the easy implementation of signal-oriented applications without considering communication delays. The basis of a *Flexible Joint Module* is an FPGA with a communication interface. The network of FPGAs connected by a communication infrastructure is the static reusable part of the platform. In contrast to that, peripheral components with dedicated interfaces and specific implementations of algorithms on the FPGAs are adapted to the application specific constraints.

The first implementation of a *Flexible Robot Platform* is realized for DLR HAND II [12]. The basis of the finger electronics is a configurable FPGA (Altera Flex10k) that interfaces the finger torque and position sensors as well as three motor controllers. The Joint Modules are connected to a host controller CPU by an implementation of the IEEE1355, a simple packet oriented communication protocol.

The latest design based on the *Flexible Platform* concept is a 7DoF light-weight robot for medical surgery [13]. The module consists of a Xilinx VirtexIIPro Platform FPGA that interfaces the entire joint periphery, e.g. position and torque sensors, motor and brake. Furthermore, the FPGA provides configurable computing power: configurable logic, internal

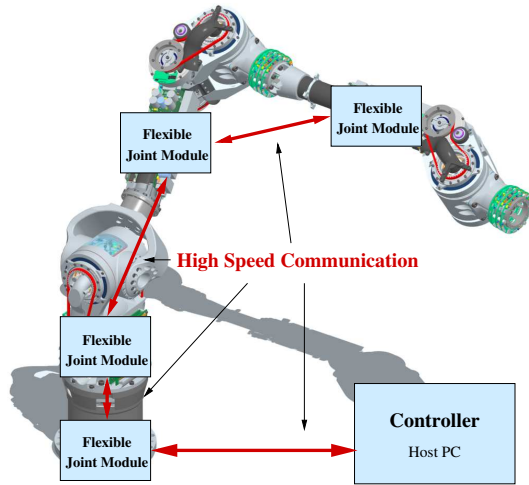


Fig. 10. The Robot Platform

PowerPC, and dedicated multipliers. With that, the motor current control and a joint state observer is implemented. Using the integrated high speed links, the FPGA implements a communication infrastructure derived from the SpaceWire Protocol [14]. High bandwidth (1 GBit/sec) and low latency ($< 50\mu s$) help to reduce the influences of communication delays (see Fig. 9).

VI. A SURGICAL ROBOT BUILT WITH THE FLEXIBLE PLATFORM

The surgical robot is built from four Flexible Joint Modules: one single joint and three coupled joint modules constitute the 7DOF robot (see Fig. 10). The joint motors' current control loop is implemented on that modules. So, the HAL developer has to integrate that remote joint functionality and present a complete current controlled robot to application developers.

Fig. 11 depicts the actor model of the robot control application. The application, i.e. the outer control loop, sees only the HAL interface. The HAL integrates the remote joint functionality and represents a complete current controlled robot. Therefore, all functionality has to be implemented that

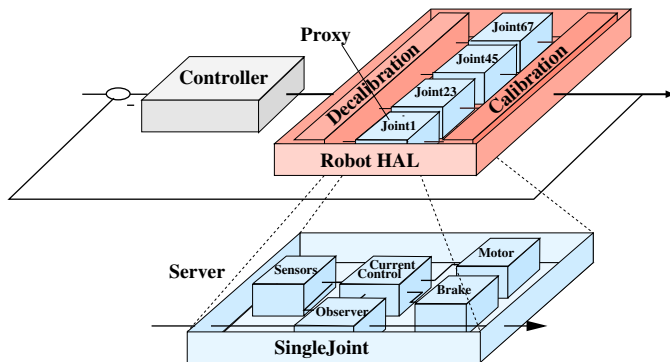


Fig. 11. Robot Control Application with HAL

is not provided by the hardware but necessary for a proper HAL, such as sensor value calibration. On the other side, the hardware developer has to implement the current controller on the flexible joint module.

The specification of the joint module interfaces with SFMiddleware IDL by the hardware developer is the first step in this process. So, the HAL implementation can be built upon this stable interface, even so no hardware is available yet. HAL and hardware development can be executed concurrently.

Fig. 12 depicts module Joint1 as actor. The corresponding IDL for this module is listed in Lst. 1. Note, how the input and output signals of Fig. 12 are combined to one eventtype *Desired* and *Actual*, respectively. This is valid, because the application's model of computation is SDF, and it defines all signals to fire synchronously. The *MotorControl* interface provides the attributes of the joint's current controller.

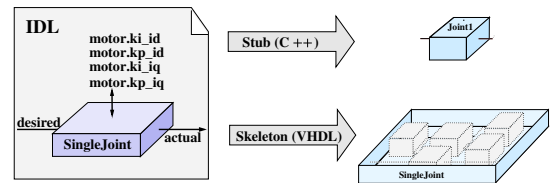


Fig. 12. Joint1: Generate Stub/Skeleton From IDL

On the client side, the Stubs, i.e. joint proxies, are generated in C++ from the IDL descriptions. The HAL developer uses those proxies to implement the robot's hardware abstraction. The IDL represents the hardware functionality exactly as is. This also concerns the signal data types. For example, the position sensor signal is described as 16bit fixed point value, exactly as implemented on the FPGA. An application developer does not want to be faced with such details. So the HAL for the surgical robot implements sensor value calibration and decalibration and presents all sensor values as floating point SI-values (see Fig.11).

On the server side, Skeletons are generated as VHDL code from the IDL descriptions. A Skeleton presents the module interface as VHDL signals to the hardware developer. So, the current controller implementation can be directly connected to that VHDL signals.

SFMiddleware efficiently connects joint proxies and remote joint module implementation with its application specific Real-Time Signal Broker (see Fig. 13). Therefore the Generic Transactions were implemented with the SpaceWire protocol. So, a call of the consume function in the Stub of Joint1 by the HAL is transmitted as SpaceWire packet to the joint module and results in the corresponding VHDL signal in the FPGA implementation. The static application specification of SFMiddleware allows an optimized implementation of that process. For the surgical robot, we achieved the desired current control loop cycle time of 3 KHz. All this is hidden by SFMiddleware. So, neither HAL developer nor hardware developer is faced with infrastructure. Both develop simply against the interfaces formally specified with the IDL.

Listing 1. IDL for Joint1

```

uses motor;
uses observer;

module robot
{
  component SingleJoint
  {
    struct Sensors {
      bfixed<16,15> torque[2];
      bfixed<16,15> position;
    };

    eventtype Actual {
      observer::State jointState;
      Sensors sensors;
      motor::State motorState;
    };

    eventtype Desired {
      boolean enable;
      motor::Current current;
    };

    interface MotorControl {
      attribute bfixed<16,17> ki_id;
      attribute bfixed<16,12> kp_id;
      attribute bfixed<16,17> ki_iq;
      attribute bfixed<16,12> kp_iq;
      readonly bint<8> temperature;
    };

    publishes Actual actual;
    consumes Desired desired;

    provides MotorControl motor;
  };
};

```

VII. CONCLUSIONS

The concept of a flexible robot platform, consisting of transparently connected flexible joint modules, together with a signal-oriented hardware abstraction keeps the impact of changing hardware specifications as local as possible. This is achieved by the decoupling of component communication and functionality, introduced by actor-oriented design and retained throughout implementation by *SFMiddleware*. The signal-oriented middleware allows a control application to reach into the specification of hardware implementation and vice versa. The formal specification of component interfaces and the signal-oriented communication model of actors enables the automatic generation of an application specific communication infrastructure. Thus, the design effort to create a valid hardware abstraction (i.e. software driver) is scaled down and rapid prototyping of complex robotic systems is rendered possible. Moreover, the static system specification approach used in *SFMiddleware* enables compile-time optimization, i.e. a small run-time footprint. Thus, we achieve control cycles of more than 10 kHz.

Further work includes the implementation of a *SFMiddleware* IDL Compiler for C++, VHDL and SystemC. We plan to integrate *SFMiddleware* with Simulink/Real Time Workshop.

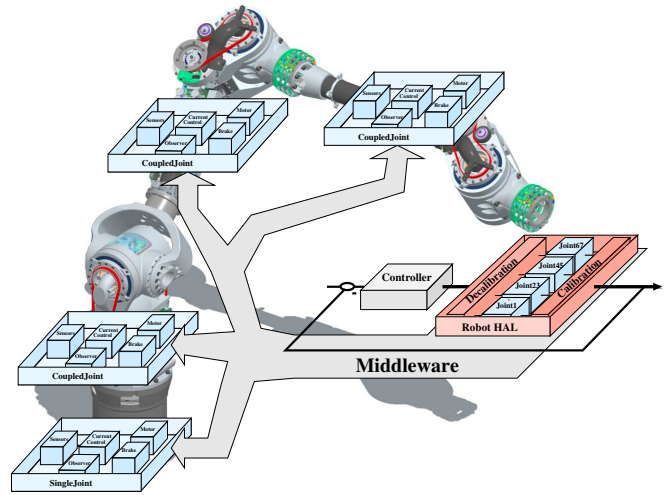


Fig. 13. The Surgical Robot Implementation

REFERENCES

- [1] O. M. Group, *RealTime-CORBA Specification*, 2nd ed., November 2003, formal/03-11-01.
- [2] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the tao real-time object request broker," in *Computer Communications*, vol. 21, no. 4, April 1998.
- [3] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck, "Towards component-based robotics," in *Proc. International Conference on Intelligent Robots and Systems*, Edmonton, Canada, 2005, pp. 3567–3572.
- [4] K.-U. Scholl, J. Albiez, and B. Gassmann, "Mca - an expandable modular controller architecture," in *3rd Real-Time Linux Workshop*, Milano, Italy, 2001.
- [5] R. T. Vaughan, B. Gerkey, and A. Howard, "On device abstractions for portable, reusable robot code," in *Proc. International Conference on Intelligent Robots and Systems*, Las Vegas, USA, October 2003, pp. 2121–2427.
- [6] H. Bruyninckx and P. Soetens, "Generic real-time infrastructure for signal acquisition, generation and processing," in *Fourth Real-time Linux Workshop*, Boston, USA, December 2002.
- [7] A. Sangiovanni-Vincentelli, L. Carloni, F. de Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *41st Design Automation Conference*, San Diego, CA, 2004.
- [8] E. A. Lee, "Computing for embedded systems," in *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 2001, pp. 21–23.
- [9] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, "Actor-oriented design of embedded hardware and software systems," in *Journal of Circuits, Systems, and Computers*, vol. 12, 2003, pp. 231–260.
- [10] E. A. Lee and D. Messerschmitt, "Synchronous data flow," *IEEE Proceedings*, vol. 75, no. 9, November 1987.
- [11] O. M. Group, *Common Object Request Broker Architecture: Core Specification*, 3rd ed., November 2002.
- [12] S. Haidacher, J. Butterfass, M. Fischer, M. Grebenstein, K. Joehl, K. Kunze, M. Nickl, N. Seitz, and G. Hirzinger, "Dlr hand ii: Hard- and software architecture for information processing," in *Proc. IEEE International Conf. on Robotics and Automation*, Taipei, Taiwan, 2003.
- [13] T. Ortmaier, H. Weiss, U. Hagn, M. Grebenstein, M. Nickl, A. Albu-Schäffer, C. Ott, S. Jörg, R. Konietschke, and G. Hirzinger, "A hands-on-robot for accurate placement of pedicle screws," in *Proc. IEEE International Conf. on Robotics and Automation*, Orlando, USA, May 2006.
- [14] E. C. for Space Standardization (ECSS), *SpaceWire - Links, nodes, routers and networks*, 2003, eCSS E-50-12A.