

# ERSP: A Software Platform and Architecture for the Service Robotics Industry

Mario E. Munich, Jim Ostrowski, Paolo Pirjanian  
Evolution Robotics, Inc.  
Pasadena, California, USA  
Email: {mario,jim,paolo}@evolution.com

**Abstract**—In this paper we describe the need for, and the characteristics of, a software architecture for commercial robotic products. We describe the Evolution Robotics Software Platform (ERSP™), which provides a commercial-grade software architecture for mobile robots. The architecture has been designed to be modular, scalable, lightweight, portable, and reusable. It follows a hybrid model of data flow, combining behavior-based processing modules for real-time reactions with event-based task planning routines. We provide a detailed description of the main architectural components and some basic usage studies. We also highlight areas where compromises have been made and for which continuing work is needed.

**Index Terms**—Software Platform, Software Architecture, Service Robotics Industry.

## I. INTRODUCTION

Products such as Sony’s entertainment robot Aibo™, Electrolux’s robotic vacuum cleaner Trilobite™, iRobot’s Roomba™, and prototypes from numerous robotic research laboratories across the world foreshadow a rapid development of service robots over the next several years. In fact, the momentum created by these early products combined with recent advances in computing and sensor technology has laid the groundwork for the creation of a major industry for service robots.

The applications for service robotics range from existing product categories, where advanced robotic technologies are incorporated to enhance and improve the product functionality, to new products, where robotic technologies enable and inspire completely new functionalities. One example of the first case is vacuum cleaners, where the enhancement is reflected in an increased degree of autonomy. Examples of the second case are companionship robots and robots for interactive entertainment.

A necessary element to make the industry for service robots grow is to develop core robotic technologies, both hardware and software, that deliver flexibility and autonomy at low cost. This will allow the robotics community, especially the commercial sector, to focus on the value-added applications development rather than solving the core robotic problems. We have focused much of our research and development on such low cost software and hardware components for vision, navigation, and architecture. We describe here our development of an architecture for the service robotics

industry. This paper is organized as follows. Section II summarizes the previous work on robot architectures. Section III describes the desired characteristics of a robotic architecture. Sections IV-VII present the Evolution Robotics Software Architecture (ERSA™). Section VIII presents case and usage studies of ERSA with respect to the desired characteristics. Finally, Section IX establishes some concluding remarks.

## II. PREVIOUS WORK

To motivate the current exposition of an architecture for commercial robotics, it is useful to review previous work on architectures from both the academic and commercial robotics sectors. In the 1960s the AI and robotics communities developed *symbolic planners*, such as STRIPS used to control Shakey the SRI robot [5]. Later it was realized that pure planning approaches, also known as “sense-plan-act” architectures, suffer when faced with the dynamics and uncertainty of the real world.

Realizing the limitations of planning systems a new approach was taken in the mid 1980s which can be viewed as the *deliberative approach*. The deliberative architectures, such as NASA’s NASREM [4], were characterized by having a hierarchical control structure where higher level modules provided goals for lower levels. However, these systems also relied heavily on symbolic representations and hence suffered from similar problems as symbolic planners.

In the late 1980s, Brooks proposed a complete departure from using planners and symbolic representations. His Subsumption Architecture [3] relied on reactive modules that implement robot competencies by reacting to sensory data without much processing. Brooks demonstrated that subsumptive robots could react to real time events in the environment and exhibit very robust behaviors. The reactive approach later evolved into the behavior-based approach [2], where robot control is distributed among goal-oriented modules known as behaviors.

The late 1980s and the 1990s saw new approaches which attempted to combine the best of deliberative and reactive approaches into the hybrid or 3-layer architecture [7]. The hybrid, deliberative-reactive, architectures generally consist of a reactive executive that deals with real-time responses to dynamic events, a high-level deliberator that reasons about

long term goals, and a mediator that coordinates the interaction between the two layers. Examples of hybrid systems include the Task Control Architecture [12], ATLANTIS [6], and 3 Tier Architecture. For a detailed overview of architectures, see [11].

Even today there are several world-wide efforts trying to develop a common software control architecture. Since NASREM, NASA has initiated two significant efforts for developing a common, software control architecture for robotics. These efforts are Mission Data Systems (MDS) and CLARAty [10] (Coupled-layer Architecture for Robot Autonomy). There are also several efforts from commercial companies working on common architectures, including Saphira and ARIA from ActivMedia, OPEN-R/SDE from Sony, RoboStudio from NEC and more.

### III. CHARACTERISTICS OF A COMMERCIAL ROBOTICS ARCHITECTURE

In this section, we will highlight the characteristics that seem most important for a commercial robotics software architecture. In doing so, we seek to derive guidance from the significant research literature on this topic, as well as motivation from the constraints and requirements of the service robotics sector.

In a broad view, we will define a robot as a system that has a number of sensors, processing units (e.g., computer), and actuators. The software architecture must provide an appropriate abstraction of the hardware components and a solid paradigm for decision making and action under uncertainty and for real-world applications. By real-world we mean unstructured, usually cluttered, dynamic, and typically unknown environments. Further, the software architecture must enable the programmer or system engineer to rapidly develop and customize the control software towards a specific target, whether it is a vacuum cleaner, an entertainment toy, or a home companion.

We focus here on software architectures designed explicitly to support the following commercial sectors of robotics: Entertainment, Toys, Companions, Household Appliances, Retail Enterprises, and possibly Industrial sectors. Each sector will have different and possibly conflicting requirements and constraints— thus tradeoffs must be made to realize the final implementation of the architecture.

In a long term view, the architecture used for each sector would have an implementation adapted for that particular sector. This would embody the “Niche Finding” property referred to by Arkin [1], whereby architectures must find their place in a competitive world by being adapted to their particular domain area, or niche. For instance, the entertainment sector would require the architecture to support powerful models of personality and emotion, which most likely is an unnecessary capability for industrial robotics. We also note that the ability to tailor the architecture to satisfy

a particular commercial sector can be critically important, since the computational platform for robots ranging from industrial robots to toys can vary significantly from Pentium CPUs with gigabytes of memory and gigahertz clock speeds down to embedded CPUs running at tens of megahertz with only kilobytes of RAM.

The following list outlines the main characteristics and requirements for the design and implementation of a software architecture for commercial robotics. It is clear that not all of the characteristics can be satisfied simultaneously— some are even contradictory— but our goal is to seek balances that optimize the tradeoffs.

- **Modularity:** The architecture should include support and guidelines for a modular code structure that allows only the applicable portions of the code to be installed, executed, or updated. This includes providing the support infrastructure for easily composing the modules to form a seamlessly integrated system.
- **Code reusability:** This implies that modules can be used in a variety of applications and should support working across different configurations, for example, as sensor type and placement is modified.
- **Portability and platform independence:** In many cases, it is desirable to use the software developed for an application across different hardware (e.g., CPUs, robots) and software (e.g., operating systems) platforms. On the hardware side, this means that the software should be easily configurable for different robot configurations, such as sensor type and layout, motor type, and overall mobility. It is desirable that the software can be easily ported to run on different CPU architectures and different operating systems, such as Linux, Windows™, or MacOS™. In the commercial sector, an important aspect of this is to support implementations that can run on embedded microprocessors with limited memory and processing power.
- **Scalability:** It should be easy to expand the system by adding new software modules and hardware components. Also, the overhead of supporting modular components should increase reasonably with the scale of the system.
- **Lightweight:** The modularity and flexibility provided by the architecture should not introduce significant overhead to the system. This is clearly a conflicting goal with many of the other characteristics; however, the architecture must provide the right balance between generality and efficiency. This is especially important for commercial robotics, where computational overhead directly impacts cost.
- **Open and flexible:** The system should provide access to the architecture implementation through a well-established API, and should provide flexibility in allowing customizations that respect the overall architecture design.
- **Dynamic reconfigurability:** It should allow for dynamic reconfiguration of the system, including adding, removing, upgrading, or reconnecting components to the system.

- **Ease of integration with external applications:** Although this can be hard to measure, the goal should be to provide convenient and flexible mechanisms to integrate the modules developed under this architecture with external libraries and applications, for example, customized software developed by third parties.

- **Networking support:** As networking infrastructure becomes more and more commonplace, it is important that the architecture properly support working across Ethernet networks. Furthermore, remote process control and shared data across networks is desirable.

- **Fault monitoring:** The system should support component level determination of task success or failure, along with mechanisms for handling failures at different levels of the architecture. It should also provide self-monitoring through online evaluation of its state, as well as satisfaction of its task objectives.

- **Testing infrastructure:** The architecture should support the ability to test each component, as well as provide system-level testing facilities, both at the development and product stages.

- **Reactive and deliberative:** The system should provide tight perception-action feedback loops to react promptly to unexpected situations as well as higher level planning for efficient use of resources over longer time frames. Plans should guide, not control, reactive components.

There are also several other items that are linked more to a particular implementation of an architecture than the overall high-level characteristics of an architecture include. Some of these include the support of adaptation and learning, robustness in the face of uncertainty and/or incomplete information, and the use of standard software development practices, such as documentation, interfaces, and maintainability.

In addition, we feel strongly that the *tools* provided to support working within an architecture can be just as important as the components that are supported by the architecture. These tools include, for example, tools for rapidly building applications, configuring the system, debugging during development, or visualizing the system and analyzing its performance. Finally, it is important that an architecture provides a framework and guidelines to support and instruct code development. For example, given the system task, resource limitations, etc. the framework should guide one to efficiently design, implement, and evaluate a system.

#### IV. EVOLUTION ROBOTICS SOFTWARE ARCHITECTURE

In this section, and the three following sections, we introduce our vision and implementation of an architecture that attempts to satisfy the above characteristics. The *Evolution Robotics Software Platform* (ERSP™) provides basic components and tools for rapid development and prototyping of robotics applications. The software architecture, called the *Evolution Robotics Software Architecture* (ERSA™), is the

underlying infrastructure and one of the main components of ERSF.

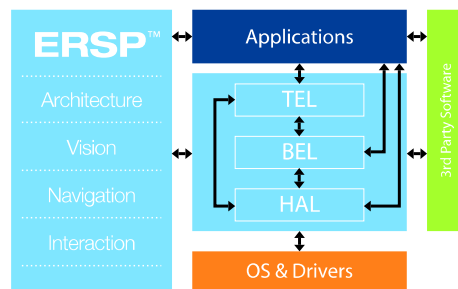


Fig. 1. ERSF structure and relation to application development.

Figure 1 shows a diagram of the software structure and the relationship among the software, operating system, and applications. There are five main blocks in the diagram – three of them, *Applications*, *OS & Drivers*, and *3rd Party Software* correspond to external components to ERSF. The other two blocks correspond to subsets of ERSF: the core libraries (left-hand-side block) and the implementation libraries (center block). In this paper, we focus on the *Architecture* (ERSA) portion of the core libraries. The other core libraries provide additional components for robotic applications. The *Vision* component includes algorithms for color segmentation and tracking, optical flow computation, and *Visual Pattern Recognition* (ViPR™) module [9]. The *Navigation* component includes exploration, mapping, obstacle avoidance, path planning, and *visual Simultaneous Localization and Mapping* (vSLAM™) [8]. The *Human-robot Interaction* module includes voice recognition, speech synthesis, and tools for building GUIs.

The ViPR system provides state-of-the-art algorithms for recognition of visual patterns using inexpensive camera hardware and limited computing power. The system is invariant to rotation and affine transformations of the patterns and to changes in scale and lighting, and robust to occlusions. The vSLAM algorithm enables low-cost, vision-based mapping and navigation in cluttered and populated environments. No initial map is required, and it gracefully handles dynamic changes in the environment, such as lighting changes, moving objects, and/or people.

ERSA provides a set of APIs for integration of the different software modules and with the robot hardware. ERSF allows for building task-achieving modules that make decisions and control the robot, for orchestrating coordination and execution of these modules, and for controlling access to system resources. ERSF is composed of three layers, the *Hardware Abstraction Layer* (HAL), the *Behavior Execution Layer* (BEL), and the *Task Execution Layer* (TEL).

The architecture corresponds to a mixed architecture in which the two first layers follow a behavior-based philosophy [2], [3] and the third layer incorporates a deliberative stage for planning and sequencing [7]. The first layer, HAL,

provides *interfaces* to the hardware devices and low-level operating system (OS) dependencies. This layer assures portability of ERSA and application programs to other robots and computing environments. It also enables rapid configuration of the software to support new robot platforms or sensor layouts. The second layer, BEL, provides infrastructure for development of modular robotic competencies, known as *behaviors*, for achieving tasks with a tight feedback loop such as following a trajectory, tracking a person, avoiding an object, etc. Behaviors are the basic, reusable building blocks on which robotic applications are built. BEL also provides techniques for coordination of the activities of behaviors, for conflict resolution (action-selection mechanisms), and for resource scheduling. Finally, the third layer, TEL, provides infrastructure for developing *event-driven tasks* along with mechanisms for the coordination of task executions. Tasks can run in sequence or in parallel, and execution of tasks can be triggered by user-defined events.

While behaviors are highly reactive, and are appropriate for creating robust control loops, tasks are a way to express higher-level execution knowledge and to coordinate the actions of behaviors. Tasks can run asynchronously using event triggers or synchronously with other tasks. Time-critical modules with tight feedback loops for controlling the actions of the robot according to perceptual stimuli, such as obstacle avoidance, are typically implemented in the BEL. TEL is more appropriate to deal with complex control flow which depends on context and certain conditions that can arise asynchronously. Behaviors tend to be synchronous and highly data driven. Tasks tend to be asynchronous and highly event-driven.

## V. HARDWARE ABSTRACTION LAYER (HAL)

The Hardware Abstraction Layer (HAL) is the interface between robotics applications and the underlying hardware. HAL software controls the robot's interactions with the physical world and with low-level operating system (OS) dependencies. HAL receives physical input from sensors, such as cameras and range sensors, and provides controls to effectors that change the state of the robot's environment, for example by moving the robot or picking up objects.

HAL abstracts away the details of particular hardware devices and of platform-specific ways of interacting with hardware or other resources. We define a *resource* to be a physical device, connection point, or any other means through which the software interacts with the external environment. Resources include sensors and actuators, network interfaces, microphones and speech recognition systems, or a battery. We have also extended the notion of resources to include fundamental computational units that operate on sensory data. For example, both vSLAM and ViPR can be accessed directly as resources.

The software module that provides access to a resource is

termed a *resource or device driver*. A driver implementation uses the appropriate operating system or other library function calls to interact with the underlying resource. A driver implements resource functions and is, by definition, dependent on the operating system and device specifics. Resource drivers communicate with hardware devices through a communication bus. The description of the resources, devices, busses, their specifications, and the corresponding drivers are managed through configuration files based on the *Extensible Markup Language* (XML).

To protect higher-level modules from these dependencies, HAL provides a number of well-defined *interfaces* for interacting with a variety of robotic devices. These interfaces are a set of public, C++ abstract classes that buffer the user from the details of a given implementation and facilitate the interaction with resources using real world concepts and units. The particular driver implementation of these interfaces is determined at run-time, based on the set of hardware or other resources currently being used.

For example, the HAL provides an `IRangeSensor` resource interface with methods that determine the distance to an obstacle. In addition, the `IRangeSensor` has knowledge about the uncertainty associated with its measurements. An obstacle avoidance algorithm can use `IRangeSensor` to determine the position of obstacles to avoid. You can implement `IRangeSensor` for IR sensors and sonar sensors. At the application level, you work only with `IRangeSensor` and do not have to worry about the any device-specific details such as converting the IR or sonar readings into proper distances. The device-specific properties of the sensor are specified in an XML-based resource schema file. For an `IRangeSensor`, these might include the calibration curve that maps raw sensor values to distances, as well as a parameters describing the sensor measurement uncertainty. HAL selects the proper driver to handle these details, either for IR sensors or sonar sensors or both, depending on which is installed on the current robot. This way, algorithms can be developed generically to work with a variety of robotics platforms. The usage of interfaces for isolation of implementation details is also employed in the case of operating system-dependent constructs like multi-threading, synchronization, and file handling.

Using this system, diverse resources can access identical interfaces. A single resource driver can support multiple interfaces. For example, there can be an `IDriveSystem` resource interface that provides basic locomotion capability for simple drive systems, such as differential drive. Omnidirectional drives systems have more capabilities. One can develop a new resource interface, for example `IOmniDriveSystem`, to encapsulate these additional capabilities. The driver for the omnidirectional drive can implement both interfaces to support the basic locomotion of `IDriveSystem`. The driver can be used by components

that only need the simpler interface, but it can also support the full capability of the omnidirectional drive made available by `IOmniDriveSystem`.

A resource must be located, instantiated, and activated before it becomes available to the system. After it is no longer needed, or the system shuts down, the resource must be released and its memory reallocated. For this purpose, we utilize a **Resource Manager** that is responsible for managing the system resources across their life cycle. For the Resource Manager to load resources correctly, it must be told which resources are available and where to find them. This is achieved through the use of a **resource configuration** file— a single XML file that defines all of the robot geometry parameters (size and shape of the robot, position and orientation of sensors, etc.) and points to the correct drivers to be used for each of the resources appropriate to the given platform. As already mentioned, device drivers implement clearly defined interfaces so that the high level programs do not need to get entangled into the details of the driver implementation. For example, a program can obtain images from a camera by means of the `ICamera` interface. This abstraction isolates the application from having to deal with the fact that the camera is using a `LinuxCamera` or a `DirectX` (Windows) implementation. The following is a simple declaration of the camera device in the resource configuration for a robot running Linux (more complex declarations could include camera parameters such as image size, resolution, etc.):

```
<Device id="camera0" type="Evolution.LinuxCamera">
```

The Resource Manager loads the resource configuration file on start-up of an application, creating a list of devices for the corresponding robot. In this example, there will be a device called “camera0” of type “Evolution.LinuxCamera”. The manager decodes the type and loads the corresponding library that implements the resource driver, in this case the library containing the `LinuxCamera` class that provides the implementation of the `ICamera` interface within the namespace “Evolution”. In Windows, the type of the device would be “Evolution.DirectX”.

## VI. BEHAVIOR EXECUTION LAYER (BEL)

The Behavior Execution Layer (BEL) of ERSP is a framework for building autonomous robotics systems. Applications use the BEL to acquire sensory input, make decisions based on that input, and take appropriate actions. The fundamental building block of the BEL is the **behavior**, defined as a computational unit that maps a set of inputs to a set of outputs. This definition generalizes the classical idea from behavior-based robotics: making all behaviors support an identical interface for consistency and maximal interoperability. Behaviors cover a wide range of the robot’s functions, from driving sensors and actuators to mathematical operators, algorithms, and state machines.

Behavior inputs and outputs are pushed over **ports**. Each output port can have connections to an arbitrary number of

input ports. A port is characterized by its data type, data size, and semantic type, indicating the structure and use of the data passing through that port. These port attributes determine the validity of connections, because the start and end ports of connections must have matching types. For example, a text string output from a speech recognizer cannot connect to a port expecting an array of range sensor data. Ports and their descriptions are made available to the external user by an XML schema file associated with each behavior. In addition to ports, each behavior can have user-settable **parameters**, that are given default values in the behavior schema.

Chains of connected behaviors form **behavior networks**. ERSP executes all behaviors in the same network sequentially and at the same rate. In this sequential model, behavior execution occurs in two stages: the behavior first receives data from its input connections, and then it computes and pushes its output. For a behavior to operate on the most current input in a given network cycle, the behaviors from which it receives input must be executed first. Accordingly, ERSP uses a partial ordering so that if behavior A’s outputs are connected to behavior B’s inputs, then A executes before B. In cases where information loops occur, the user can specify which data must be backpropagated through the network. Otherwise, this coordination of behaviors and data flow is transparent to the user.

Behaviors and the behavior network infrastructure are implemented as C++ objects, and behavior networks can be implemented in C++ files. However, it is easier and more standardized to specify a behavior network to run from an XML file, by specifying the behaviors that exist in the network and the connections between input and output ports.

We use a **Behavior Manager** to load and execute behavior networks. Multiple networks can be run in parallel and at different rates. Each behavior library provides a “factory” function for each behavior type, which the Behavior Manager can call to create a new instance of that type. The Behavior Manager also reads parameters from the XML-based behavior network, which can be used at run-time to override default parameters for each behavior. Finally, during execution, the Behavior Manager coordinates the transfer of data between behaviors, according to the connectivity of the behavior network.

BEL also provides the infrastructure for aggregating behaviors into a single, meta behavior. These behavior aggregates implement the original behavior interface and can be used as any other behavior in applications. Aggregates help solve the problem of scalability of behavior networks, making the networks manageable as the number of behaviors components grows.

## VII. TASK EXECUTION LAYER (TEL)

The Task Execution Layer (TEL) provides a high level, task-oriented and event-driven method of programming an

autonomous robot. By sequencing and combining tasks using functional composition, you can create a flexible plan for a robot to execute, while writing code in a traditional procedural style. TEL support for parallel execution, task communication, and synchronization allows for the development of plans that can be successfully executed in a dynamic environment. TEL also provides a high level, task-oriented interface to the Behavior Execution Layer.

While behaviors are highly reactive, and are appropriate for creating robust control loops, tasks express higher-level execution knowledge and coordinate the actions of multiple behaviors or behavior networks. For example, an action that is best written as a behavior would be a robot using vision to approach a recognized object. An action that is more appropriate for a task, on the other hand, would be a robot navigating to the kitchen, finding a bottle of beer, and picking it up.

In addition to enabling event-driven, task-oriented processes, TEL has the added advantages of providing *familiarity* and *ease of scripting*. Defining tasks is similar to writing standard C++ functions, not writing finite state machines. Also, creating an interface with most scripting languages, including Python, is simple, allowing one to make use of the power of high level scripting languages, while controlling tasks in a natural way.

Data transfer in the TEL is done through a publish/subscription mechanism, whereby tasks can send messages to one another as *events*. Events are asynchronously broadcast from one task and can be received by multiple tasks. They consist of a type and a set of *properties*, which are name/value pairs where the name is a string and the value is a well-defined type. Events are most often conditions or predicates defined on values of variables within BEL or TEL. Complex events can be defined by logical expressions of basic events.

While tasks can be written in C++, it is also useful to link behaviors and tasks. To do this, we allow the definition of *task primitives*, which act essentially as wrappers around behavior networks, making them look and act like tasks. Other tasks can then use the task primitive exactly in the same way that they use any other task. At its core, then, a primitive is an XML file describing the connections between each behavior. The task then provides connections to the inputs of desired behaviors, and can read data from outputs of the behaviors within the network. The task primitive can then connect incoming events to inputs on behavior ports, and can route outgoing data from the behavior network to trigger events. Finally, the task primitive must handle any initialization that occurs when it is started, and any cleanup from being terminated.

### VIII. USE-CASES STUDY

This section analyzes the different characteristics of ERSR with the help of a set of use cases. Our goal is to describe

the ways in which ERSR satisfies (or falls short of) the list put forth in Section IV, and discuss tradeoffs that have been made.

#### A. Modularity

ERSR has been designed in a *modular* fashion, with each piece of major functionality being broken into a module that has minimal dependence (only as required) on other modules. Each user of ERSR decides which module of the architecture will be used in his application. For example, the ViPR module of ERSR has been successfully integrated as a stand-alone component into the latest software release for the Sony AIBO™ (ERS-7™). ViPR is used by AIBO to localize and navigate to its charging station, allowing for complete autonomy of the robot. ViPR is also used for robust command-and-control human-robotic interaction by using a set of cards which the user shows to the robot in order to initiate commands.

The integration of the ViPR module of ERSR into the Sony AIBO software only involved a custom compilation of the module for AIBO's operating system (Aperios™) and processor (64-bit RISC processor). The application software developed by Sony makes calls to the public API—no modifications of ERSR were required for integration.

A different example of modularity is given by the implementation of vSLAM in an embedded board. The amount of computation required by vSLAM made it impossible to implement the whole system on a reasonably priced processor while still maintaining a minimum execution rate. This problem was solved by off-loading the ViPR computations to a digital signal processor (DSP). This rather dramatic change in the overall design of the system (shifting from everything running on a Pentium-based, 2.4 GHz CPU to a 400 MHz PMC Sierra embedded board with DSP coprocessor) did not have an important impact on the software. The modularity of ERSR allowed us to replace the implementation of the ViPR system, supported as a resource in HAL, with a new driver that would take care of the interaction with the DSP. The interface for ViPR was unchanged, so all other software components of the system continued working seamlessly. In addition, switching between using the DSP coprocessor and using the main CPU to do the computations requires only modifications to the XML-based resource configuration file—no compilation is required. This allows rapid testing of the two systems, both for debugging and for comparing performance.

#### B. Platform Independence

ERSR has been designed to be *platform independent*, both in terms of *robot hardware* and in terms of the *CPU* and *operating system*. A set of clear interfaces and platform-independent system calls are used to abstract away any platform dependence. We cannot overemphasize the value derived from this level of abstraction. It makes shifting

between robots and computing platforms, as well as making smaller tweaks to the robot configuration or other parameters, almost trivial. This approach has proven to be successful in allowing ERSP to run on Intel x86-based machines, on two embedded boards (one based on a PMC Sierra 64-bit RISC processor and another based on an AMD Alchemy processor), and on the Sony AIBO. In addition, ERSP runs on both Windows and Linux. The main drawback that we have experienced, particularly for supporting multiple OS's, is the additional effort required to perform quality assurance testing for multiple platforms. The benefit, however, translates into reduced effort required by the end-user in working on multiple platforms.

An interesting use case that showcases platform independence of ERSP is the demonstration program that we provide for vSLAM. In this demo, the robot explores its surroundings while making a map of its environment. The demo has a client application, the Navigation Tool, that graphically displays the robot's whereabouts as well as the map being generated. This application obtains information from a server that is running within the robot program. This demonstration runs properly on an x86-based robot running either Windows or Linux. The robot server program also runs on the RISC processor-based embedded board. The robot application does not have to be changed to run in any of these platforms. It is exactly the same; only the XML resource configuration file changes!

One of the challenges of providing cross-platform functionality comes in developing tools that make use of external packages. Examples of this include GUI developer libraries (we originally worked with open-source wxWindows, but switched to the commercial QT as much easier to develop under) and third party speech libraries (we have used IBM's ViaVoice and CMU's Sphinx).

### C. Portability to Different Robot Platforms

ERSP has been designed to allow portability of applications across different robot platforms. ERSP applications are currently running on a variety of robots (shown in Figure 2). These robot platforms represent a wide range of characteristics in terms of mechanics, sensor types, geometrical configuration, and sensor parameters (intrinsic and extrinsic). The Evolution Robotics robots use a set of plastic wheels while the Pioneer uses a set of rubber tires, each with different sizes and geometric placement. The range sensors of the Pioneer are sonar sensors while the Evolution Robotics robots use IR sensors. The configuration of sensors varies a great degree among all the robots as well as the actual geometric shape of the robot. ERSP handles all these variations in graceful manner by means of the HAL layer.

As described in Section V, each robot as well as the simulator has a resource configuration file that describes the geometry of the robot and the location, type, and parameters

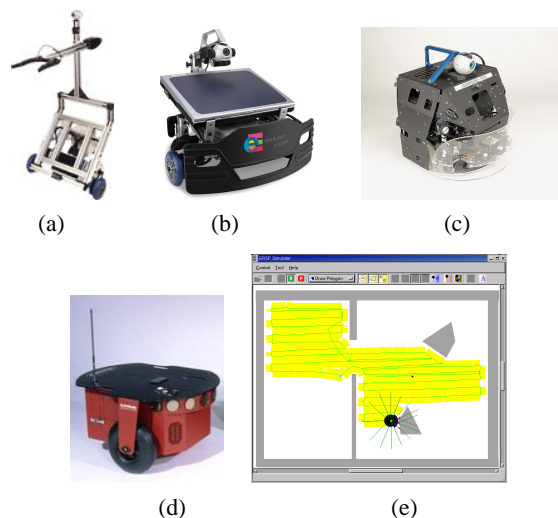


Fig. 2. A sampling of robot platforms on which ERSP runs: (a) the ER-1, (b) Scorpion, and (c) R3R robots from Evolution Robotics, (d) the Pioneer from ActiveMedia, and (e) a robot simulator from Evolution Robotics. The same application, e.g. the systematic coverage program displayed on the simulator, can be executed in all these robots by changing only a single XML configuration file (although the vSLAM module is not currently being simulated).

of the sensor. This file is loaded at run-time, and allows modification of the corresponding parameters for the desired sensors. Note that only the libraries specified in the configuration file are loaded (dynamically), which helps satisfy the *lightweight* requirement of the system.

### D. Usability Factors

The development of robotic applications is complicated by the number of elements that compose a robot. ERSP provides a number of tools, modules, and a framework aimed towards *easing application development* efforts.

XML files are used for configuration of robots, for description of the internals of resources and behaviors, and for detailing behavior networks. The use of a single, XML-based resource configuration file enhances *usability* by concentrating all system- and platform-dependent changes to a single file. Modifications of the number, type, or location of sensors or actuators thus involve only a single modification of the resource configuration file, not a re-compilation of the application. Settings for resource and behavior parameters are also described with XML files, allowing for quick modification of parameters while tuning applications. From the user standpoint, quick and easy access to parameters is key for *fast debugging* of algorithms.

An important *development tool* provided in ERSP is the Behavior Composer (shown in Fig. 3). This graphical tool allows the user to create behavior networks with a drag-and-drop procedure. A behavior is just a box that has inputs, outputs, and parameters. The behavior composer allows one to place behaviors in a network, to select the flow of data by drawing connections between the input/output ports of the behaviors, and to modify parameters with a property

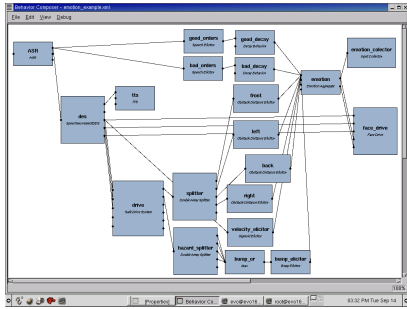


Fig. 3. Behavior composer. ERSP includes a graphical user interface tool for ease of development of applications (behavior networks) at the BEL layer. This behavior network shows an application in which the robot drives around avoiding obstacles while responding to user commands recognized with an Automatic Speech Recognition (ASR) engine.

editor. The network is saved in XML format and ready to be executed. In addition, the Behavior Composer enforces *type safety* for data flowing between behaviors, by only allowing connections between ports of compatible semantic type.

Networking support is provided within behavior networks with the *Malleable behavior*. This type of behavior opens a socket connection on a given port and handles data communication encapsulated in an XML text format. One current limitation of ERSP is that it only supports the transfer of simple data structures, e.g., strings and arrays. Transfer of more complex data structures, such as images, is handled by customized behaviors.

### E. Code Re-use and Scalability for Large Projects

ERSP has been used in several large, ongoing development efforts, especially in demonstrating different technologies. One such project, that started as a simple demonstration of vSLAM, has evolved to include automated exploration and mapping, path planning, coverage sweeps (e.g., for vacuum cleaning), and docking.

During the course of this project, the *modularity* of ERSP has allowed us to continually add on components, which can be developed, refined, and tested independently of each other. We have also had good success with *code reusability*, both within the evolution of this project, and as we (and our customers) have utilized the modules in other parallel efforts. One of the main challenges that we have faced in working with such modular code is to properly enable *scaling* of the system. The use of Behavior Aggregates has been one way to ameliorate this problem by composing modules according to their functional relationship (e.g., the `SafeDriveSystem` behavior shown on Fig. 3 is composed of approximately 15 sub-behaviors). This helps keep the number and complexity of the modules relatively constant, by abstracting the functionality as more components are added. Although the current implementation has been done in BEL, we could alternatively utilize task primitives to capture the grouping of such functionality.

Another attribute that has been demonstrated in this use-case is that of *portability*, as we have been able to run this application under both Linux and Windows, and on all of the robots shown in Figure 2 (the only exception is docking on the ER-1, since it does not have hardware to support this). The use of interfaces has allowed us to support a wide variety of sensors (range and bump sensors in the ER robots, ultrasound only in the Pioneer) and other important hardware changes, such as the complex differences found when docking.

## IX. DISCUSSION AND CONCLUDING REMARKS

In this paper, we have presented characteristics that we perceive to be important for a software architecture in the service robotics industry. We have presented the Evolution Robotics Software Platform in relation to solving these challenges. ERSP provides basic components and tools for rapid development and prototyping of robotics applications. The design of ERSP is modular and allows applications to use only the required portions of the architecture. It is designed in a way that allows the developer to reconfigure the hardware without rewriting more than a very small amount of code. ERSP also satisfies to a large extent the desire to have platform independence— independence from the specifics of the robot hardware, its sensors and actuators, its geometry, and the computing platform on which it is to run.

## REFERENCES

- [1] R. C. Arkin. Just what is a robot architecture anyway? Turing equivalency versus organizing principles. In *AAAI Spring Symposium: Lessons Learned from Implemented Software Architectures for Physical Agents*, 1995.
- [2] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.
- [3] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 1986.
- [4] J. Albus et al. NASA/NBS standard reference model for telerobot control system architecture (NASREM). Technical Note 1235, NIST, Gaithersburg, MD, July 1987.
- [5] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189–208, 1971.
- [6] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. *AAAI-92 Robot Navigation*, pages 809–815, 1992.
- [7] E. Gat. On three-layer architectures. In D. Kortenkamp et al., editor, *AI and Mobile Robots*. AAAI Press, 1998.
- [8] N. Karlsson, E. Di Bernardo, J. Ostrowski, L. Goncalves, P. Pirjanian, and M.E. Munich. The vSLAM algorithm for robust localization and mapping. In *Proc. of Int. Conf. on Robotics and Automation (ICRA)*, 2005.
- [9] N. Karlsson, M. E. Munich, L. Goncalves, J. Ostrowski, E. Di Bernardo, and P. Pirjanian. Core technologies for service robotics. In *Proc. of Int. Conf. on Intelligent Robots and Systems (IROS)*, October 2004.
- [10] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and W.S. Kim T. Estlin. CLARAty: An architecture for reusable robotic software. In *SPIE Aerosense Conference*, 2003.
- [11] P. Pirjanian. Behavior coordination mechanisms - state-of-the-art. Technical Report IRIS-99-375, Institute for Robotics and Intelligent Systems, University of Southern California, October 1999.
- [12] Reid G. Simmons. Structured control for autonomous robots. *IEEE Trans. on Robotics and Automation*, 10(1):34–43, February 1994.