Context

With the increasing intelligence and complexity of robotic systems, there is a need to both leverage working solutions and manage the rising complexity of highly capable robots. In addition to building reusable hardware components, there is a growing need to build reusable software components for heterogeneous robots. The different requirements of such systems make this proposition quite challenging.

There are three primary reasons why reusing robotic software is difficult. Taking a top-down approach, one encounters several challenges:

1. **Different categories and capabilities**: One can categorize robotic systems along different axes. One such axis can be based on their primary objective: e.g. surface exploration robots, excavation robots, construction robots, assembly robots, and so on. Another axis can be along their sensing/perceiving modalities. A third can be along their mechanisms types which define their physical abilities.

   Even within a single category, robots can have different capabilities. Consider a mechanism classification that identifies platforms as: wheeled, legged, or hybrid of the two. There are different capabilities among wheeled vehicles. Some are fully-steerable (omni-directional), others are partially-steerable (car-like Ackermann steering), while others are skid-steerable. Such capabilities determine the types of maneuvers a vehicle can support. For example, a fully-steerable vehicle can crab while a skid-steerable vehicle simply cannot.

2. **Different architectures**: Even with systems that have identical physical and sensor configurations, they can have different control and computation architectures. Control systems can be configured in different ways. One approach is to move the controls as much as possible to software implemented on the central processor keeping the communication and hardware to a minimum. The other end of the spectrum is to move as much of the computation out to distributed processors. Different combinations have different pros and cons for motion and sensor control and processing.

In addition to these top down challenges, there are several challenges associated with a bottom-up approach. In a bottom-up approach, one can look at the various constituents of a robotic system. Central to any robotic system is the mechanism that is comprised of bodies connected by different joints. Joints have different types and may be passive or actuated. Actuated joints are generally controlled by motors with feedback sensors. Sensors for robots usually include articulation sensors (encoders and potentiometers), tactile sensors of various types (digital/analog inputs), force sensors (digital/analog inputs), visual sensors (cameras, stereo pairs, and laser scanners), and inertial sensors (gyroscopes, accelerometers, and inertial measurement units). One can develop abstractions for these various components and build higher-level abstractions using these basic units. The primary challenge with this bottom-up approach again is the variation in technologies and capabilities of various actuators and sensors. Such devices from various
Vendors not only have different signal qualities, but operate in different modes which lead to software implementations that are hardware specific. For example, some motor controller only support velocity control modes while others can support position and velocity control modes. Cameras can come as either analog units with framegrabbers or as digital units on a single bus. These cameras have different capabilities for continuous image acquisition and synchronization. Nevertheless, to the robot, a motor actuates a joint, a camera acquires an image, and a stereo pair acquires synchronized images. The implementation details, while important, will vary across implementations. But abstracting these devices properly will lead to a more general and interoperable implementation of software without losing the advantages of the hardware. Extending these abstractions is necessary to support full hardware capabilities.

Using both a top-down and a bottom-up approach, one may develop and evolve various abstractions, components, standardized messages and interfaces to capture the common themes and behaviors across components and systems. One may choose a particular domain within the vast robotic discipline to study, analyze, prototype, implement, learn, and iterate. Forum such as this workshop enable the dissemination and sharing of the learned experience in an attempt towards reaching a common ground.

**Problem**

To readily deploy algorithms on various robotic platforms, algorithms must be designed independent of the implementation details of their host platforms. Because platforms have different hardware architectures, robotic software must be able to adjust the level of abstraction at which it interfaces to hardware. Adjusting the level of the interface will also enable interfacing with simulations that have different levels of fidelity.

There is a need for a framework to host many algorithms that have been developed from generalized theories in robotics. Many existing point solutions can be expressed using more general constraints to support a larger set of applications. The framework will need to support interfaces to various robotic hardware and simulation platforms.

**Forces**

- To provide a framework to host robotic algorithms that can be generalized to a number of applications.
- To provide an extendible approach that enables the overriding of generic capabilities with specific algorithms targeted for a given configuration or platform.
- To enable comparison of various components in a system. A breakdown of algorithms into functional components enables the validation and comparison of various components against others without changing the entire software base.
- To maintain efficiency and simplicity in the implementation and provide a system that is maintainable across many deployments. Generalizing an implementation introduces a level of complexity.
- To access and reason about system state at various levels of abstraction.
- To support different architectures for control and computations ranging from centralized software-driven to distributed hardware-driven where the nodes perform significant processing.
- To interface with various robotic deployments.
- To seamlessly migrate between real and simulated robots.
To use algorithms in prediction mode to support “what if” planning scenarios.

Solution

The robotic software must develop abstractions that separate intent of an action from its implementation. To support systems that can migrate functionality to hardware and to support various fidelity simulations, these abstractions must exist at various levels of the system granularity. Not only will this encapsulate implementation detail, it will help reduce the complexity of robotic software by providing levels of abstraction.

The abstractions must represent both physical as well as functional components of the system. Physical abstractions are necessary to support different hardware devices. Functional abstractions are necessary for two reasons: (i) to support various implementations of a given algorithm, and (ii) to support the processing of an algorithm elsewhere in a distributed system. Physical abstractions include controlled motors, input and output channels (digital and analog), cameras, stereo cameras, inertial measurement units, laser scanners, arms, locomotors, pan/tilt units, and so on. Functional abstractions include terrain analyzers, visual trackers, stereo processors, path planners, trajectory generators, and so on.

It is necessary to have a central model to handle information about the properties of the mechanism(s). At a minimum, the model will need kinematic properties but can also include dynamic and geometric properties. Geometric properties are included for collision detection and graphical displays. Handling this information centrally keeps the model information consistent, reduces duplication in the implementation, and allows for generic algorithms for kinematic computations. While general kinematic solutions have many advantages, provisions to support optimized custom solutions are also necessary.

In addition to physical and functional abstractions, we also have to provide abstractions for threads, time, and measurements. Robotic systems often run multiple concurrent threads. To deploy a generic multi-threaded algorithm on various systems, it is important to decouple the run-time model of the algorithm from the run-time model that is specific to a robotic platform. To support running with both simulated and real robots, the interface to system time has to also use a clock abstraction since simulations can run time at faster rates. Measurement abstractions are useful for holding time stamps and uncertainty information.

Within the class of mobile robot that carry various appendages and instruments, we found many common themes around which we build a framework to generalize robotic capabilities. We have studied at some length two capabilities: (i) robot navigation and obstacle avoidance in rough outdoor terrain, and (ii) tracking and placing instruments mounted on mobile robots at targets designated from a distance. To implement these two capabilities on the NASA research rovers, we had to develop generalized components that interacted with each other in complex patterns and with various system adaptations.

Funded by NASA’s Mars Technology Program, and in collaboration with NASA Ames Research Center, Carnegie Mellon, and University of Minnesota, JPL has designed, implemented, deployed, and tested CLARAty (Coupled Layer Architecture for Robotic Autonomy) on a number of robotic research platforms [1].

CLARAty provides two layers for the development of robotic software: the Decision Layer and the Functional Layer. The Decision Layer uses a model-based declarative approach to describe
the high-level activities and their constraints. Using a search engine, a plan is generated that properly orders the activities based on the mission goals and system constraints. A schedule is generated from this plan and then executed by sending goal to the Functional Layer. The Functional Layer is a set of abstractions at various levels of granularity that can support different implementations of functional algorithms. It can also be adapted at various levels to real or simulated hardware components.