

# CLARAty and Challenges of Developing Interoperable Robotic Software

Issa A.D. Nesnas, Anne Wright, Max Bajracharya, Reid Simmons, Tara Estlin

Email:firstname.lastname@jpl.nasa.gov

Jet Propulsion Laboratory, California Institute of Tehcnology, Pasadena, CA 91109

March 10, 2003

## Abstract

*In this article, we will present an overview of the Coupled Layered Architecture for Robotic Autonomy. CLARAty develops a framework for generic and reusable robotic components that can be adapted to a number of heterogeneous robot platforms. It also provides a framework that will simplify the integration of new technologies and the comparison against similar technologies. CLARAty consists of two distinct layers: a Functional Layer and a Decision Layer. The Functional Layer defines the various abstractions of the system and adapts the abstract component to real or simulated devices. It provides a framework and algorithms for low- and mid-level autonomy capabilities. The Decision Layer provides the system's high-level autonomy which reasons about global resources and mission constraints. The Decision Layer interacts with the Functional Layer using a client-server model, which accesses information at multiple levels of granularity. In this paper, we will also present some of the challenges in developing interoperable software for various rover platforms.*

## 1 Introduction

Developing intelligent capabilities for robotic systems requires the integration of various technologies from different disciplines. It also requires the interaction of various software components within a real-time system, and the management of uncertainties resulting from the interaction of the robot with its environment. The uncertainties from the environment, the complexities of software/hardware interactions, and the variability of the robotic hardware make the task of developing robotic software complex, hard, and costly. However, a number of the algorithms developed for robotic systems can be generalized and applied to a number of platforms irrespective of the details of their implementations. It is such algorithms that the Coupled Layered Architecture for Robotic Autonomy (CLARAty) is trying to provide a

framework for while maintaining the ability to easily integrate platform-specific algorithms.

CLARAty is domain-specific robotic architecture designed with four main objectives: (i) to reduce the need to develop custom robotic infrastructure for every research effort, (ii) to simplify the integration of new technologies onto existing systems, (iii) to tightly couple declarative and procedural-based developments, and (iv) to operate a number of heterogeneous rover with different physical capabilities and hardware architectures. CLARAty is a collaborative effort among California Institute of Technology's Jet Propulsion Laboratory, Ames Research Center, Carnegie Mellon University, and a number of other universities and members from the robotics community.

## 2 Background

With the increased interest in developing rovers for future Mars exploration missions, a significant number of rover platforms have been designed and built over the past decade. Several NASA centers and university partners use these platforms to test their newly developed technologies to improve the autonomous robot capabilities. Because of the differences in the mechanical and electrical designs of these vehicles, they shared little in terms of software infrastructure. Transferring capabilities from one rover to another has been a major and costly endeavor. Because robotics systems cover several domain areas, researchers of a single domain also needed to integrate their newly developed technology into the complex robotic environment. Proper integration requires an in-depth understanding and characterization of the behavior of various components of the system, which may vary from one platform to another.

One of our goals is to provide a design that allows researchers to use various components spanning domains outside their immediate expertise but have these components flexible and extendible to support various applications. To do so, we need to capture well-understood and well-developed knowledge from the various domains into generalized and reusable domain components. Just like an operating system provides a level of abstraction from the computational hardware, our goal is to provide a level of abstraction from the robotic hardware implementation that will allow developers to

“integrate once and run anywhere.” Of course, there are physical limitations to this goal due to the large variability among rover capabilities.

The development of robotics and autonomy architectures dated back several decades. We will not attempt to provide a comprehensive review of the body of work upon which this effort builds. Typical robot and autonomy architectures are comprised of three levels - Functional, Executive, and Planning levels \cite{ALAMI98} \cite{GAT98} \cite{SIMMONS98}. Some architectures emphasized one area over others and thus became more dominant in that domain. For example, some architectures emphasized the planning aspects of the system \cite{estlin:ijcai99} \cite{FIRBY89}, others emphasized the executive \cite{BORRELLY98} \cite{simmons:icirs98}, while others emphasized the functional aspects of the system \cite{mobility-software-isrobotics} \cite{nesnas-stanisic-94} \cite{schneiderchen-pardo-castellote-wang-98}. There has also been efforts that aimed at blurring the distinction between the planning and executive layers \cite{fisher:ieeaaero98} \cite{knight:spaceops00}. Other architectures did not explicitly follow this typical breakdown. Some focused on particular paradigms such as fuzzy-logic based implementations \cite{konolige-saphira-97} or behavior-based implementations \cite{arkin-89} \cite{brooks-86}. There has been considerable effort in architectures that addressed multiple and cooperating robots \cite{parker-95} \cite{mataric-97}.

One difference between the *CLARAty* architecture and the conventional three-level architectures is the explicit distinction between levels of granularity and levels of intelligence. In conventional architectures both granularity and intelligence were aligned along one axis. As you move to higher abstractions of the system, intelligence increases. This is not true for the *CLARAty* architecture, where intelligence and granularity are on two different axes. In other words, the system decomposition allows for intelligent behavior at very low levels while still maintaining the structure of the different abstraction levels. This is similar in concept to hybrid reactive and deliberative systems.

### 3 An Overview of the CLARAty Architecture

The CLARAty architecture has two distinct layers: the Functional Layer and the Decision Layer. The Functional Layer uses an object-oriented system decomposition and employs a number of known design patterns \cite{gamma} to achieve reusable and extendible components. These components define an interface and provide basic system functionality that can be adapted to

real or simulated robots. It provides both low- and mid-level autonomy capabilities. The Decision Layer couples the planning and execution system. It globally reasons about the intended goals, system resources, and state of the system and its environment. The Decision Layer uses a declarative-based model while the Functional Layer uses a procedural-based model. Because the Functional Layer provides an adaptation to a physical or simulated system, specific model information is collocated in the system adaptations. The Decision layer receives this information by querying the Functional Layer for predicted resource usage, state updates, and model information. However, additional adaptation specific heuristics are often used with current planners to assist in plan generation. These adaptation specific heuristics, which are only used by the Decision Layer, can be accessed directly and not via the Functional Layer.

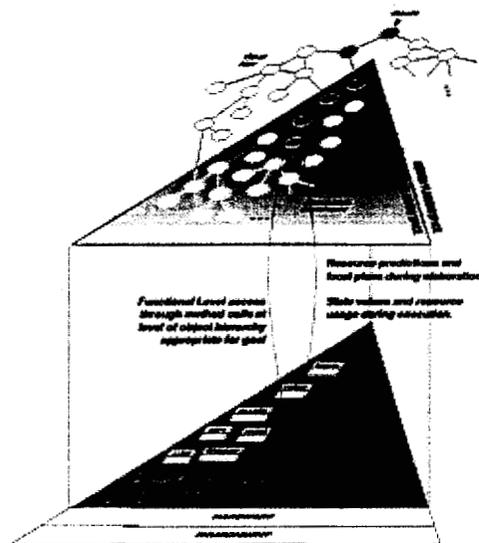


Figure 1: Decision Layer (red) interaction with the Functional Layer (blue)

The Decision Layer can access the Functional Layer at various levels of granularity. The architecture allows for overlap in the functionality of both layers. This intentional overlap allows users to push the declarative model to elaborate to lower levels of granularity while allowing the Functional Layer to provide fixed order procedures for mid-level autonomy capabilities. In the latter case, the Decision Layer serves as a monitor to the execution of the Functional Layer behavior which can be interrupted and preempted depending on mission priorities and constraints. Figure 1 shows the relationship between the Functional Layer and the Decision Layer in CLARAty.

#### 3.1 The Functional Layer

The Functional Layer includes a number of generic frameworks centered around various robotic-related disciplines. Of the packages included in the Functional Layer are: I/O, motion control and coordination, locomotion, manipulation, navigation, mapping, terrain evaluation, path planning, science analysis, estimation, simulation, and system behavior. The Functional Layer provides the system's low- and mid-level autonomy capabilities. High-level control algorithms such as vision-based navigation, sensor-based manipulation, and visual target tracking use a predefined sequence of operations and are often implemented in the Functional Layer. However, in some case, it is also possible to generate such sequence of operations by modeling them as activities in the database and have the Decision Layer schedule activities based on appropriate constraints.

The Functional Layer has four main features. First, it provides a system level decomposition with various levels of abstractions. For example, in the locomotion domain, a general locomotor provides an interface to any type of mobility platform whether it is a wheeled vehicle, a legged mechanism, or a hybrid of the two. A functional specialization of the locomotor is the wheeled locomotor. This specialized introduces the concept of wheeled mobility and wheel configuration. This abstraction extends the locomotion interface to include additional capabilities that can be accomplished. Further extension of wheel locomotor includes special type of wheel locomotor with known locomotion models.

Second, the Functional Layer separates algorithmic capabilities from system capabilities. It is important to decouple system limitations from the algorithm limitations in order to avoid propagation of assumptions that are unique to a particular platform. Algorithms are expressed in their most general terms without compromising understandability and efficiency. Where efficiency requirements are not met, specializations are provided to overwrite the general solution. An example of such capability can be found in the manipulation domain. General inverse kinematics algorithms provide a generic solution for all serial manipulators but are often not efficient. As a result, they are overwritten with specialized, more efficient, versions. The general versions however, can still be used in instances where specialized solutions have not been formulated or for validating the specialized implementation.

Third, the Functional Layer separates the behavioral definitions and interactions of the system from the implementation. This not only allows the dynamic binding of adaptations at runtime, but it also makes both the functional and implementation trees extensible. For example, a wheeled locomotor separates the interface to hardware from the specialization along the behavior and

model configurations of the system. Another example is the controlled motor which separates the specialization to a particular hardware controller from the functional specialization of a controlled motor to a joint (which extends the motor functionality by imposing checking of joint limits on all the move commands) This pattern has been implemented on various systems and is known as the bridge pattern \cite{gamma}.

Fourth, the Functional Layer provides flexible runtime models. The runtime model is part of the abstraction model, of which, one part is associated with the generic functionality and the other with the adaptation. The runtime model associated with the adaptation is dependent on particular capabilities of the underlying hardware and can change from one system to another. For example, a system with distributed motion control does not need to run the servo control threads and possibly the trajectory generation threads on the main processor. This capability can be implemented in firmware and on distributed processors.

### 3.2 The Decision Layer

The Decision Layer is a global engine that reasons about system resources and mission constraints. It includes general planners, executives, schedulers, activity databases, and rover models.

The Decision Layer plans, schedules, and executes activity plans. It also monitors the execution modifying the sequence of activities dynamically when necessary. The goal of a generic Decision Layer is to have a unified representation of activities and interfaces with the Functional Layer. The current instantiation of the Decision Layer features a tight coupling of the planner and executive which interacts with a separate Functional Layer at all levels of system granularity. The planner implementation is CASPER \cite{casper} and the executive implementation is TD \cite{tdl}.

The Decision Layer interacts with the Functional Layer using a client-server model. The Decision Layer queries the Functional Layer about availability of system resources or for predicting the usage of a particular resource for a given operation. The Decision Layer sends commands to the Functional Layer at various levels of granularity. The Decision Layer can utilize encapsulated Functional Layer capabilities with relatively high-level commands, or access lower-level functionality and combine it in ways not provided by the Functional Layer. The former is valuable when planning capabilities are limited, or when under-constrained system operation is acceptable. The latter is valuable if detailed, globally optimized, planning is possible, or if resource margins are small. CLARAty supports both modes of operation.

Status is reported from the Functional Layer to the Decision Layer asynchronously or synchronously at rates specified by the Decision Layer.

## 4 Challenges in System Decomposition

The proper decomposition for a generic robotic system, in large, depends on what elements of the software are targeted for reuse in future applications. One approach for an architectural decomposition is to highlight the runtime model and inter-component communication mechanism independent of the domain it addresses \cite{schneider-chen-pardo-castellote-wang-98}. Another would be to highlight the states of the system making them explicit with global scope \cite{mds}. A third would be to highlight the abstract behavior and interface to the states of the system while hiding runtime models. It is the latter approach that CLARAty adopted in order to hide the variability that arises from various implementations.

Two fundamental notions of CLARAty are abstractions at various levels of granularity and encapsulation of information at the appropriate levels of the hierarchy. First, abstractions are an important notion in a robotic system in order to reduce complexity and to provide an operational interface at various levels of the system architecture. Algorithmic development can occur at any level of abstraction. At any given level, higher levels can be replaced by user defined substitutes. Second, without the proper encapsulation, implementation specific information and assumptions can “bubble up” to higher levels and break reusability across domains and platforms. This does not mean that CLARAty does not support platform specific algorithms. Specific algorithms are ones that can neither be generalized nor is it effective to generalize in order to expand the scope of their applicability.

There are three main types of abstractions in the Functional Layer: (1) data structure classes, (2) generic/specialized physical classes, (3) generic/specialized functional classes. All classes are designed to maximize code reuse across disciplines, eliminate duplicated functionality without compromising efficiency, and simplify code integration.

Generic components both physical and functional: (a) provide interface definitions and implementations of basic functionality, (b) provide local executive capabilities, (c) manage local resources, and (d) support state and resource queries by the Decision Layer.

### 4.3 Data Structure Classes

Data structure classes, which handle data transformation and storage, enable easy propagation of software optimization, and allow easy serialization and transport. One characteristic of data structures is that they do not have any executive capability, making them the easiest to implement and port on multiple operating systems. While their efficiency is of prime importance, they themselves do not invoke other threads (tasks). These classes provide the extended interface for communication among generic physical and functional components. Since general-purpose data structures are reusable beyond the scope of robotics applications, we are leveraging standardized developments such as the Standard Template Library \cite{austern-stl}. However, domain specific implementations are developed. Such classes include points, bits, arrays, vectors, matrices, rotation matrices, images, homogeneous transforms, quaternions, frames, frame trees, messages, and resources.

### 4.4 Generic Physical Classes

A generic physical component (GPC) defines the structure and behavior of a physical object in an abstract sense. Some of these classes have partial implementations since they will eventually to adapted to a physical/simulation classes that will complete their implementation. A generic physical component can be extended along two axes: functional and implementation. The functional extension includes addition of control and operational capabilities. The implementation axes include

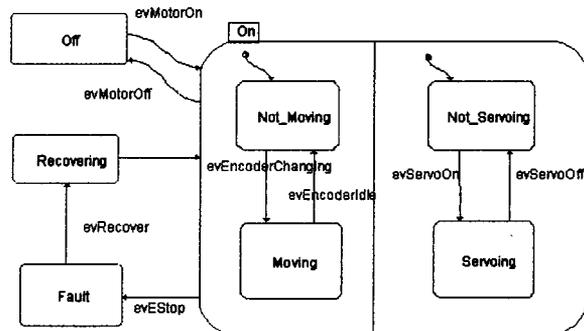


Figure 2: Parallel state machine for a generic physical controlled motor

specialization to hardware and overriding generic default implementation. A generic physical component can also have a model which describes the device without specifying how it is implemented. For example, a locomotor abstraction provides an interface to any type of mobility mechanism, whether it be wheeled, legged, or hybrid. The interface includes controlling any point on the vehicle to be moved along a path or otherwise to a

different point in the world. It also includes controlled the speed of such a maneuver. There are also a number of queries about the state of the vehicle and its pose. Without further knowledge of the type of mechanism, it is not possible to get further information without imposing additional constraints on the type.

In addition to defining the interface and behavior, the generic physical classes also define the state machines of an abstraction. Figure 2 shows the state machine for a generic controlled motor.

Generic physical classes can be active, i.e. they provide their own threading model. Examples of such components are: manipulator, locomotor, controlled motor, wheel, camera, and digital I/O to name a few. A complete list of characteristic and of these can be found at \cite{Nesnas IROS}.

The base abstraction for generic physical components is the Device class from which other classes derive. It uses a generic mechanism to query device properties and can retrieve both generic and specialized properties of a device via a generic mechanism. The Device provides a centralized infrastructure for device thread safety. Devices include three types of information: attributes (static parameters such as initialization parameters), parameters (dynamic parameters that are changed by the user or application at runtime), and device output data. Devices also carry textual information their given names and ancestries.

#### 4.5 Generic Functional Classes

A generic functional class is an abstract class that describes the interface and functionality of a generic algorithm. A generic functional class can have a complete implementation of its functionality because it interfaces with generic physical classes. Examples of generic functional classes are: Mapper, Navigator, Traversability Analyzer, Visual Tracker, and so on. Just like physical classes, functional classes are active and can generate separate threads of execution and run within multiple threads. In other words, these classes can have local executive capability.

For example, a navigator provides a functional behavior that will evaluate a terrain and assess its traversability, then move a mobility platform using both local and global information. The navigator interfaces with a locomotor for controlling the vehicle, an estimator for querying of pose information, a traversability analyzer for converting sensor data into a model of the world, an action selector to determine the appropriate next action for the robot to perform given its current state, and cost functions for converting terrain evaluation data into a form that can be used by the planner. A detailed description of the

navigator functional classes can be found in \cite{urmsen}.

The estimator is another type of generic functional component that can be specialized to a particular type of state propagation filter such as a Kalman Filter or a Bayesian Filter.

Equivalent to the device class for generic physical classes is the behavior that can be used as a base class for generic functional classes.

## 5 Specialized Physical/Functional Classes

Specialized classes are extensions of generic classes that adapt the general mechanism and configurations to a particular robotic platform or a specialized system configuration. An example of a specialized physical class is found in the Rocky 7 rover implementation. During the adaptation process of the mast software, the generic manipulator class is specialized to a Rocky 7 mast class. The manipulator class provides generic forward and inverse kinematics, joint motion control, trajectory tracking, conditional motion, and error recovery. The specialized Rocky 7 mast class specifies the link dimensions, joint limits, actuator type, and end effector type. It also overrides the generic kinematics of the manipulator class with the closed-form kinematics that are specifically derived for this type of manipulator.

A specialized functional class is a class that is derived from its generic counterpart and specializes a particular configuration. For example, a rocker bogie locomotor model is a specialization of a generic wheel locomotor model (the rocker bogie is a mechanism that has differential motion of the left and right sides of a six wheel vehicle – commonly used for Mars exploration rovers).

### 5.6 Runtime and Data Flow Models

Because CLARAty supports systems with different hardware architectures, the runtime model changes across robotic platforms. As a result, it is important to encapsulate the specialized runtime implementation but characterize the usage of resources as a result.

Two models of data flow are used in CLARAty. Both push and pull models are used depending on the adaptation layer and matching hardware architecture. For systems that have bandwidth limitations on a shared bus, where the need for the data is asynchronous and constitutes a subset of all possible information from that can be attained from the bus, then a pull model allows maximum flexibility. If the usage is predictable and

synchronous then a push model is used. For a given bus, and if both modes are supported by hardware, it is possible to switch the system between these two modes depending on the operation regime. For example, on a rover that uses a shared bus for communicating with distributed motion controllers on the mast and the arm, the system only retrieves information on the manipulator that is under control.

Generic classes can also employ a similar mechanism for data flow. However, the mechanism must be able to support an extendible data set with strong typing. A publish subscribe mechanism was design and implemented that can handle extendible data type. The mechanism registers objects with a particular data source. As information becomes available, a number of pending objects will execute their operations based on their current priority and registration order.

## 6 Implementation of Locomotion on various mobile platforms

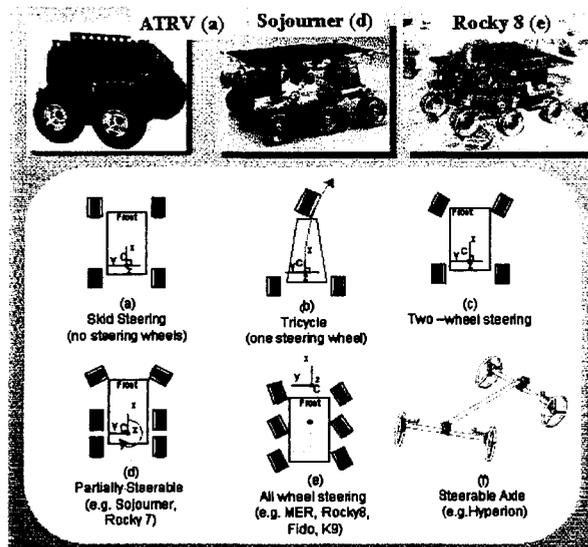


Figure 3: Various types of wheeled locomotors

One of the main challenges in developing generic components and adaptating them to different robots stems from the variability of the platforms and their capabilities. For example, surface exploration robots can be wheeled vehicles, legged mechanism, or a hybrid of the two. We will focus our discussion on wheeled locomotors. There are several challenges in developing generic classes for locomotion. The first challenge comes from the different capabilities that wheeled locomotor can exhibit depending on their configuration.

We will consider the locomotion capabilities and motion control architecture for a number of mobile platforms to which we were adapting CLARAty. Figure 3 shows the ATRV, Rocky 7, Rocky 8, FIDO, K9, Sojourner, and Hyperion rovers. While all these are wheeled vehicles, they have different maneuvering capabilities. The proper classification of these vehicles will be based on the domain knowledge of the kinematics and dynamics for controlling these vehicles. One approach which we adopted to separate vehicles as moveable axle (e.g. Hyperion) vs fixed axle (or fixed contact model - all others). For fixed axle robots, one can further classify these robots as non-steerable (or skid steerable) such as ATRVs, partially steerable such as Rocky 7 and Sojourner rover, and fully steerable such as Rocky 8, FIDO, and K9 rovers. Partially steered vehicle can have different configurations. For example the Sojourner rover which has six drive wheels has two non-steerable center wheels. On the other hand, Rocky 7 has only two steerable front wheels. As such, partially steerable wheel locomotors are constrained to instantaneously move about a rotation center that lies along the non-steerable wheel axle (or an axle that averages all non-steerable axles in order to minimize slip). Fully steerable vehicles can do crab maneuvers and can maintain a certain heading while driving along a path trajectory. Partially steerable vehicle have more constraints and use Ackermann maneuvering to compensate for crabbing

A general way for describing motion for all fixed axle models is by specifying three independent control variables that are a function of time: delta length of traverse, delta heading, and motion direction angle. For fully steered vehicles one can use all three parameters. For partially steered vehicles, the motion direction angle is fixed by the fixed axle(s). The latter is a degenerate case of the fully steered model.

A second challenge that arises in addressing these class of vehicles comes from the accessibility of the system's control parameters. For example, the ATRV can provide control for only the side of the vehicle but not for individual wheel. So the control model for the vehicle is different than others.

A third challenge steps from the different motion control architectures. Consider the motion control architecture of Rocky7, Rocky8, K9 and FIDO. While closer in resemblance to each other from, say the ATRV (all have six wheels and almost all have fully steerable capabilities), the control architecture for each vehicle is uniquely different.

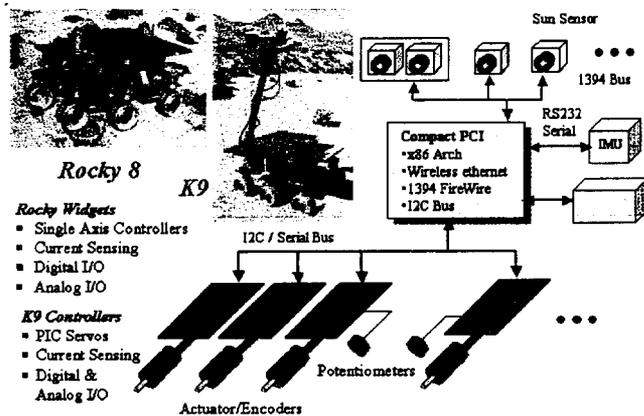


Figure 4: Distributed motion control architecture for Rocky 8 and K9

Starting with the Rocky 8 and K9 rovers (Figure 4), both rovers use a distribution control architecture where each motor interface with a single-axis microprocessor controlling the motor servo loop and sometimes trajectory profiling. Distributed microcontrollers can, as in the case of Rocky 8, also perform analog and digital I/O operations. They also possess some additional programmable processing capabilities. In a distributed system, micro controllers are connected to the main processor via some type of a serial bus. Rocky 8 uses a single I2C bus for its locomotor, arm, and mast. So architecturally, there is an important coupling between controlling the arm and locomotor simultaneously, which is support by bus bandwidth should also be supported by the software architecture. The K9 rover uses a dedicated multi-drop RS422 serial link for the locomotion motors.

Another aspect of hardware architecture is hardware synchronization. The K9 system supports hardware synchronization of motors via an electrical signal. The Rocky 8 rover implements synchronization in software by loading all motor trajectories first and then issuing start commands to all motors sequentially to minimize latency between the first and last motor. Once again the software architecture should support these two different modes of synchronizations. As such, support for device groups is an essential part of the architecture. The flexibility in implementation of group commands is also important since hardware implementations vary.

The Rocky 7 system uses COTS microcontroller chips (LM629) (Figure 5), however, they are connected to the host processor via a custom parallel port connection with chip multiplexing. While communication speeds are superior to Rocky 8 and K9 serial links, all actuators share the same bus which has to be arbitrated to support locomotion and manipulation operations. Hence there is a coupling between arm, mast, and locomotion

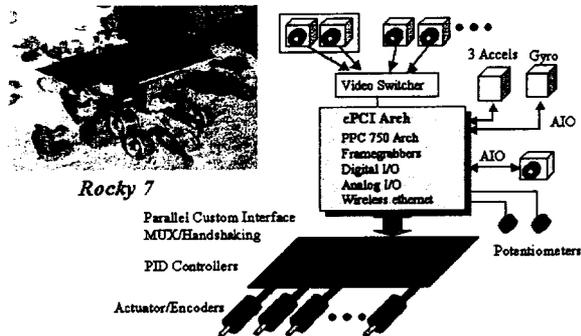


Figure 5: Custom parallel bus for centralized motion controllers on Rocky 7

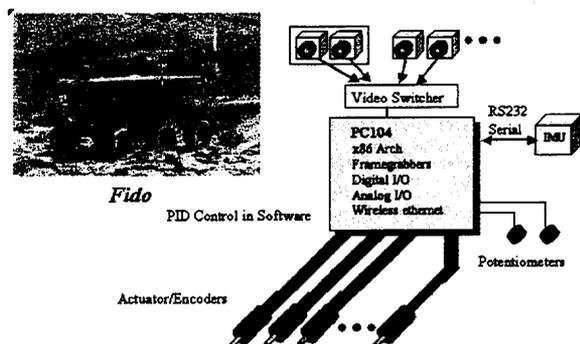


Figure 6: The FIDO motion control architecture

commanding. As in the case of Rocky 8 and K9, the closed loop motion control is done on separate non-programmable control law microcontroller.

Figure 6 shows the FIDO motion control architecture. FIDO uses a hardware mapped centralized control architecture. All motors and encoders are connected to PC104 analog output boards and quadrature encoder boards respectively. Hardware states/registers are memory mapped via a PCI bus to the host processor making them readily accessible to the software architecture. There is virtually zero cost from a software architecture standpoint to retrieve the value of any register. Hence the coupling among the various actuator/encoders states are abstracted by the hardware. However, since the host processor is the only one in the system, the servo control for all actuators has to be done on the main processor. This has the advantage of allowing the architecture to easily modify the control law and insert validation and checks in case failure of a motor or encoder occurs. In some cases that cannot be achieved with COTS processors. On the other hand, this places a requirement on the operating system and the software architecture to be able to achieve hard real-time control to service all servo loops. It also introduces a coupling between the servo loops and the applications algorithms which are competing for the same

resources. So while K9 and Rocky 8 can operate in a soft real-time environment such as Linux, the FIDO rover requires the operating system and supporting architecture to run in hard real-time.

To run effectively all the above platform, CLARAty had to be able to support the various hardware architectural models. For a system such as Rocky 8, pushing all motor and I/O information via the I2C bus to the main processor limits the bandwidth since the type of information requested by maybe different depending on the algorithm that is operational at any instance in time. So a pull model is used in this case. However, cooperative scheduling is used to acquire the needed information for motion control. In comparison, this information is readily available on the FIDO rover. The Rocky 7 and K9 rovers are hybrids of these two examples.

Despite the variations, there is a level of abstraction that can be used to interoperate across these systems. For this example, the controlled motor and motor group abstractions are used. Given that each motor is controlled via the generic controlled motor interface, the runtime model for each implementation will vary. For instance, the FIDO motor runs two threads, one for closed loop PID servo control, and a second at a lower rate for trajectory generation. The Rocky 7 and K9 motors run no additional threads and passes the necessary trajectory parameters to the firmware which will runs its own hardware thread. The Rocky 8 motor will use a single thread for trajectory generation while the microcontroller runs the PID control law. To the user of a controlled motor, the abstraction of the controlled motor and the resources its adaptation consumes is what is needed without necessarily exposing the details of the implementation. The controlled motor abstraction is then used in a wheel abstraction and later a wheel locomotor model.

While these are three different implementations of a motion control system, the behavior and functional requirements of the controlled motor are the same. In any of these implementation, you would still like to do position commanding, velocity profiling, and trajectory control. You would also like to detect and report stall conditions and be able to interrupt the motion. You would also like to read the current and desired positions, velocities, accelerations, and health status. For a person developing vision-based navigation component for a mobile robot, it is only necessary to understand the behavior of the component rather than be required to have intimate knowledge of the implementation and hardware details. Nor should they have a particular implementation inadvertently influence their design of vision-based navigation algorithms. The motor and coordinated motors classes are an abstract representation for motion control that define what the components are supposed to do.

These components hide the details of the implementation without compromising particular features of the hardware.

Some preliminary results showed that for one implementation of wheeled locomotor, about 90% in terms of lines of code that was reusable among FIDO, Rocky 8 and Rocky 7. For the controlled motor part, the reusable percentage ranged from 50%-70%. These statistics considered that the drivers, although generic to run in different environments, are still deemed too specific to include in the reusable count. (these statistics are rough estimates of line counts which include comments and spacing)

## 7 Summary

Currently, the CLARAty architecture has been adapted to five real rovers with different hardware architectures and physical capabilities. It has also been adapted to the ROAMS high-fidelity simulation. CLARAty is operating the Rocky 8, FIDO and Rocky 7 rovers at JPL. It is also running on the K9 rover at ARC and an ATRV rover at CMU. Various capabilities have been demonstrated on various vehicles.

We have presented a brief overview of the CLARAty architecture and some of the design trades and reasons for their adoption. We have also presented some of the challenges that are encountered working and adapting an architecture on various systems.

We are continuing the development of CLARAty to achieve its goals of a generic reusable robotic software base that we hope to publish as open source.

## 9 Acknowledgments

The work described in this paper was carried out by the entire CLARAty team at the Jet Propulsion Laboratory, California Institute of Technology, under a contract to the National Aeronautics and Space Administration, and at Carnegie Mellon University and Ames Research Center.

## 8 References:

- [1] R. Alami et al. An Architecture for Autonomy. *International Journal of Robotics Research*, 17(4), April 1998.
- [2] Ronald C. Arkin. Motor schema based mbilt robot navigation. *Int'l Journal of Robotics Research*, 4(8):92-112, 1989.

- [3] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*.
- [4] Addison-Wesley Professional Computing Series, Reading, MA, October 1998.
- [5] J. Borrelly et al. The ORCCAD Architecture. *International Journal of Robotics Research*, 17(4), April 1998.
- [6] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Transactions on Robotics and Automation*, 2(1):14–23, 1986.
- [7] Bruce Powel Douglass. *Real-Time UML - Developing Efficient Objects for Embedded Systems*. Addison-Wesley Longman, Inc., Reading, MA, December 1998.
- [8] Tara Estlin, Gregg Rabideau, Darren Mutz, and Steve Chien. "Using continuous planning techniques to coordinate multiple rovers." In *Proceedings of the IJCAI99 Workshop on Scheduling and Planning meet Real-time Monitoring in a Dynamic and Uncertain World*, Stockholm, Sweden, August 1999.
- [9] I.A.D. Nesnas, R. Volpe, T. Estlin, H. Das, R. Petras D. Mutz, "Toward Developing Reusable Software Components for Robotic Applications" *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, Maui Hawaii, Oct. 29 - Nov. 3 2001
- [10] R. Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, Department of Computer Science, 1989.
- [11] Forest Fisher, Steve Chien, Leslie Paal, Emily Law, Nassar Golshan, and Michael Stockett. An automated deep space communications station. In *Proceedings of the 1998 IEEE Aerospace Conference*, Aspen, CO, March 1998.
- [12] E. Gat. On Three-Layer Architectures. In D. Kortenkamp, R. Bonnasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots*, Boston, MA, 1998. MIT Press.
- [13] R. Knight, S. Chien, T. Starbird, K. Gostelow, and R. Keller. Integrating model-based artificial intelligence planning with procedural elaboration for onboard spacecraft. In *Proceedings of Space Ops 2000*, Toulouse, France, June 2000.
- [14] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti. The saphira architecture: A design for autonomy. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1):215–235, 1997.
- [15] Maja J. Mataric. Behavior-based control: Examples from navigation, learning, and group behavior. *Journal of Experimental and Theoretical Artificial Intelligence*, 2–3(9):232–336, 1997.
- [16] I.A. Nesnas and M.M. Stanišić. A robotic software developed using object-oriented design. In *ASME Design Automation Conference*, Minnesota, 1994.
- [17] Lynn Parker. Alliance: An architecture for fault tolerant multi-robot cooperation. In *ORNL TM12920*, Oak Ridge National Laboratory, Oak Ridge, TN, 1995.
- [18] G. Pardo-Castellote S. Schneider, V. Chen and H. Wang. Controlshell: A software architecture for complex electromechanical systems. *Int'l Journal of Robotics Research*, 17(4), April 1988.
- [19] R. Simmons and D. Apfelbaum. A Task Description Language for Robot Control. In *IEEE/RSJ Intelligent Robotics and Systems Conference*, Vancouver Canada, October 1998.
- [20] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Proceedings of the International Conference on Intelligent Robots and Systems*, Vancouver, Canada, October 1998.
- [21] Mobility Software. <http://isrobotics.com/rwi/software.htm>. Real World Interface, a division of IRobot, Somerville, MA.
- [22] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The clarity architecture for robotic autonomy. In *Proceedings of the 2001 IEEE Aerospace Conference*, Big Sky, Montana, March 2001.