CLARAty and Challenges of Developing Interoperable Robotic Software

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

*Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109
**NASA Ames Research Center, Moffet Field, Sunnyvale, CA 95134
***Carnegie Mellon University, Pittsburg, PA 15214

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 enable the comparison of various elements. 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 components to real or simulated devices. It provides a framework and the algorithms for low- and mid-level autonomy. The Decision Layer provides the system’s high-level autonomy, which reasons about global resources and mission constraints. The Decision Layer accesses information from the Functional Layer at multiple levels of granularity. In this article, 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. Hence, it has become increasingly important to leverage robotic developments across projects and platforms. Because a number of the algorithms developed for robotic systems can be generalized, it is possible to use these algorithms on various platforms irrespective of the details of their implementations. It is such algorithms that the Coupled Layered Architecture for Robotic Autonomy (CLARAty) [22] is trying to provide a framework for, while maintaining the ability to easily integrate platform-specific algorithms.

CLARAty is a 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 algorithms, and (iv) to operate a number of heterogeneous rovers with different physical capabilities and hardware architectures. CLARAty is a collaborative effort among several institutions: 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 [18][22]. Several NASA centers and university partners use these platforms to test their newly developed technologies in order to improve the autonomous robot capabilities. Because of isolated software development efforts, exacerbated by differences in the mechanical and electrical designs of these vehicles, they have historically shared little in terms of software infrastructure. As a result, transferring capabilities from one rover to another has been a major and costly endeavor. Furthermore, 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 be 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 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 that result from the large variability in rover capabilities.
The development of robotics and autonomy architectures dates 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 [1][11][19]. Some architectures emphasized one area over the others and thus became more dominant in that domain. For example, some architectures emphasized the planning aspects of the system [7][8], others emphasized the executive [4][19], while others emphasized the functional aspects of the system [20][16]. There have also been efforts that aimed at blurring the distinction between the planning and executive layers [9]. Other architectures did not explicitly follow this typical breakdown. Some focused on particular paradigms such as fuzzy-logic based implementations [12] or behavior-based implementations [2][5]. There has also been considerable effort put in architectures that addressed multiple and cooperating robots [15][23].

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 CLARAty, 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 some 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 [10] to achieve reusable and extendible components. These components define an interface and provide basic system functionality that can be adapted to a variety of 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, all 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.

The Decision Layer accesses the Functional Layer at various levels of granularity (Figure 1). The architecture allows for overlap in the functionality of both layers. This intentional overlap allows users to elaborate the declarative model to lower levels of granularity. But it also allows the Functional Layer to build higher level abstractions (e.g. navigator) that provide 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.

3.1 The Functional Layer

The Functional Layer includes a number of generic frameworks centered on various robotic-related disciplines. Packages included in the Functional Layer are: digital and analog I/O, motion control and coordination, locomotion, manipulation, vision, 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. Control algorithms such as vision-based navigation, sensor-based manipulation, and visual target tracking that use a predefined sequence of operations are often implemented in the Functional Layer. In some cases though, it is possible to generate such sequence of operations by...
modeling them as activities and having the Decision Layer schedule instantiations of these activities based on appropriate mission goals and constraints.

The Functional Layer has four main features. First, it provides a system level decomposition with various levels of abstractions. For example, 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 specialization introduces the concept of wheeled mobility and wheel configuration. This functional specialization extends the locomotion interface to include additional capabilities. Further extensions of the wheeled locomotor include special types of wheel locomotors with known locomotion models.

Second, the Functional Layer separates algorithmic capabilities from system capabilities. It is important to decouple system limitations from the algorithmic 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 this 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, are useful in instances where the specialized solutions have not been derived yet or for validating the specialized implementation.

Third, the Functional Layer separates the behavioral definitions and interactions of the system from the implementation. This separation 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 considerations related to the behavioral and functional models from considerations related to the hardware interface. 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 is used in various parts of the architecture and is known as the bridge pattern [10].

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 a distributed motion control architecture does not need to run the servo control and trajectory generation threads on the main processor. This capability can be implemented in firmware 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, executors, schedulers, activity databases, and rover and planner specific heuristics.

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. The current instantiation of the Decision Layer which we use at JPL features a tight coupling of the planner and the executor. For this example, the planner implementation is the CASPER planning and scheduling system [7] and the executor implementation is the TDL executive system [19].

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 in order to predict the resource usage of 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 capabilities and combine them 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 on resources, state conditions, and activity execution 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 [16]. Another would be to highlight the states of the system making them explicit with global scope [6]. A third would be to highlight the abstract behavior and interface to the states
of the system while hiding runtime models. CLARA adopted the latter approach in order to hide the variability that arises from various implementations.

Two fundamental notions of CLARA are: (1) abstractions at various levels of granularity, and (2) 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. 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 CLARA does not support platform specific algorithms. Specific algorithms are ones that either cannot be generalized, or would be ineffective if generalized to a broader scope.

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.

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

4.1 Data Structure Classes
Data structure classes, which handle data transformation and storage, enable easy propagation of software optimization, and allow easy serialization and transport between processors. One characteristic of data structures is that they do not have any executive capability. While their efficiency is of prime importance, they themselves do not invoke other threads. These classes provide the extended interface for communication among generic physical and functional components. Since general-purpose data structures are reusable across the scope of robotics applications, we are leveraging standardized developments such as the Standard Template Library [3]. However, not all such needs could be adequately met from standardized sources. CLARA provides some general data structures and a number of domain-specific ones. Such classes include points, bits, arrays, vectors, matrices, rotation matrices, images, homogeneous transforms, quaternions, frames, frame trees, messages, and resources.

4.2 Generic Physical Classes
Generic physical components (GPC) define the structure and behavior of physical objects in an abstract sense. Some of these classes have partial implementations since specialized physical or simulation classes will complete their implementation. A generic physical component can be extended along two axes: function and implementation. The functional extension includes the addition of control and operational capabilities. The implementation axis includes specialization to hardware and, where necessary, the overriding of the generic default implementation. A generic physical component can also have a model that 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 allows specifying a point on the vehicle to be moved to a different point in the world, and allows other parameters, such as the path and speed, to either be specified or left unconstrained. There are also a number of queries about the state of the vehicle and it pose. Without further knowledge of the type of mechanism, it is not possible to get more information without imposing additional constraints on the type. In addition to defining the interface and behavior, the generic physical classes also define the finite state machines of an abstraction.

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 I/O to name few. A complete list of these components and their characteristics can be found in [13].

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 class 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 have standardized interfaces to query their given names and ancestries.

4.3 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, and visual tracker. Just like physical classes, functional classes can be active and can generate separate threads of execution.

An example of a generic functional class is the navigator. The 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 path
planner. A detailed description of the navigator functional
classes can be found in [21].

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.

4.4 Specialized Physical/Functional Classes

Specialized classes are extensions of generic classes that
adapt the general configuration or algorithm to a
particular robotic platform. An example of a specialized
physical class is found in the Rocky 7 rover
implementation. For the development of the Rocky 7 mast
software, the generic manipulator class is specialized to a
Rocky 7 mast class. This class specifies the link
dimensions, joint limits, actuator types, and end effector(s).
The base manipulator class provides the
generic forward and inverse kinematics, joint motion
control, trajectory tracking, conditional motion, and error
recovery. The Rocky 7 mast class overrides the generic
kinematics of the manipulator class with the closed-form
kinematics that are specifically derived for the Rocky 7
mast.

Specialized functional classes are derived from their
generic counterparts. They specialize a particular
configuration and tune the behavior. 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 rovers).

4.5 Runtime and Data Flow Models

Because CLARA 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.

Two models of data flow are used in CLARA: 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, and
where the need for the data is asynchronous and
constitutes a subset of all possible information that can be
obtained, 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 system configuration.
For example, on a rover that uses a shared bus for
communicating with distributed motion controllers
connected to both the mast and the arm, the system may
only retrieve information on the manipulator that is being
controlled.

Generic interfaces bridge between the timing
requirements of consumers and actual data flow of a
given device, as well as support extendible data sets with
strong typing. Consumers can choose whether to force a
new update, access stored data from the most recent
transactions, or retrieve a data source object. In the latter
case, the consumer can customize its timing constraints,
and either use it for future queries or pass it on to another
consumer such as a data logger. When new information
becomes available, any consumers waiting on such a data
source wake up and receive the update. If new data is not
available within the timing constraints of a given
consumer, they wake up empty handed and can choose to
force an update.

5 Implementation of Locomotion on various
mobile platforms

One of the main challenges in developing generic
components and adapting them to different robots stems
from the variability of the platforms and their capabilities.
In this section, we will use the example of wheeled
locomotion to illustrate how to use domain knowledge to
classify vehicles to enable the development of generic and
reusable classes. We will also discuss the challenges that
arise from adapting the generic algorithms to a number of
rover platforms with different hardware architectures.

Wheeled locomotors have different capabilities depending
on their mechanical configuration. Consider the
locomotion capabilities of a number of mobile platforms
shown in Figure 2 (the ATRV, Rocky 7, Rocky 8, FIDO,
K9, Sojourner, and Hyperion rovers). These wheeled
vehicles 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, is to separate vehicles with moveable axles (e.g.
Hyperion) from ones with all fixed axles (or fixed contact
model - all others). For fixed axle robots, one can further
classify these as non-steerable (or skid steerable) such as
the ATRVs, partially steerable such as the Rocky 7 and
Sojourner rovers, and fully steerable such as the Rocky 8,
FIDO, and K9 rovers. Partially steerable vehicles can
have different configurations. For example the Sojourner
rover has six drive wheels and two non-steerable center
wheels. On the other hand, Rocky 7 has only two
steerable front wheels. As such, partially steerable wheeled locomotors are constrained to instantaneously move about a rotation center that lies along the non-
steerable wheel axle (or a virtual 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 vehicles have more constraints and cannot independently control path and
heading, but can use parallel a parking maneuver to achieve a crab equivalent [14].

A general way for describing the motion of 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 constrained by the fixed axle(s). The latter is a
degenerate case of the fully steered model.

A second challenge that arises in addressing these classes
of vehicles comes from the accessibility to the system’s
control parameters. For example, the ATRV provides
independent control for each side of the vehicle only but
not for each individual wheel. So the control model for
the vehicle is different from those vehicles where each
wheel can be controlled individually.

A third challenge stems from the different motion control
architectures. Consider the motion control architecture of
Rocky 7, Rocky 8, K9 and FIDO (all have six wheels and
almost all have full steering capabilities). While closer in
resemblance to each other than to the ATRVs, for
instance, the control architecture for each vehicle is still
unique. Starting with the Rocky 8 and K9 rovers [17]
(Figure 3), both rovers use a distributed motion control
architecture where each motor interacts with a single-
axis microprocessor controlling the motor servo loop and,
in some cases, profiling a trajectory. 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, microcontrollers are
connected to the main processor via some type of a serial
bus. The K9 rover uses a multi-drop RS422 serial link for
the control of its mobility motors. Rocky 8 uses a single
I2C bus for its locomotor, arm, and mast subsystems.
There is an important coupling between the arm/mast
and the locomotor as a result of the shared bus. The software
architecture has to enable the simultaneous operation of
the manipulator and locomotor subsystems by managing
the shared resource. While the two subsystems are linked
in their implementation, functionally they are not.

Another aspect of hardware architecture is hardware
synchronization. The K9 system supports hardware
synchronization of motors via broadcasting serial
commands which tell all axes to synchronously execute
their loaded trajectory, or synchronously stop. 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 CLARAty architecture. The flexibility
in the implementation of group commands is also
important since hardware implementations can vary dramatically.

The Rocky 7 system uses commercial-of-the-shelf (COTS) microcontroller chips (LM629) for the motor control (Figure 4). These controllers are laid out on a central motion control board and are connected to the host processor via a custom parallel port connection with chip multiplexing. All actuators in the system share the same bus, but the communication bandwidth is higher than the serial links for both Rocky 8 and K9. Similar to Rocky 8, this motion control board supports the locomotion and manipulation subsystems. As in the case of Rocky 8 and K9, the closed loop servo control is done on the microcontrollers that have fixed control law with programmable parameters and modes.

Figure 5 shows a third implementation of a motion control architecture. The FIDO rover [18] uses a centralized hardware-mapped control architecture. The motors are directly connected to an analog output board and the encoders are directly connected to a quadrature encoder board. All hardware states and registers from the PC104+ boards are mapped via the PCI backplane to the host processor's memory making them readily accessible to the software. There is virtually zero cost from a software architecture standpoint to retrieve the value of any register as compared to the other systems. Hence the coupling among the various motor/encoder states is abstracted by the hardware. However, since there is only one processor (host) in the system, the servo loops for all actuators have to be done on the main processor. This introduces a coupling between the servo control of the motors and the application algorithms which will be competing for the same computational resources. It also places a requirement on the operating system and the software architecture to meet hard real-time scheduling guarantees. So while the K9 and Rocky 8 rovers 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. On the other hand, the FIDO architecture has the advantage of allowing the software to easily modify the control law and insert validation checks in case a motor or encoder failure occurs.

Despite all these architectural variations, there is a level of abstraction that can be used to interoperate across these systems. Given that each motor is controlled via the generic controlled motor interface, the runtime model for each implementation will vary. For a system such as Rocky 8, pushing all motor and I/O information via the I2C bus limits the bandwidth since the type of information requested may vary depending on the algorithm that is in operation at any time. Using a pull model, a single thread for trajectory generation (20 Hz) is used on the host while the microcontrollers run a lead/lag compensator for servo control. Motor commands are sent to the hardware using a synchronous cooperative scheduling model. Alternatively, the FIDO rover uses two threads, one for closed loop PID servo control at 200 Hz, and a second for trajectory generation. The Rocky 7 and K9 motors run no additional threads and pass the necessary trajectory parameters to the motor controllers, which run their own hardware threads. Hence, an asynchronous communication model is used.

While these are four different implementations of a motion control system, the behavior and functional requirements of the controlled motor are the same. All these implementation variations are part of the encapsulated specialization of the controlled motor and motor group abstractions. 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. In any of these implementations, 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. Furthermore, you would like to read the current and desired positions, velocities, accelerations, and health status. For a person developing a general wheel coordination algorithm for a mobile robot, it should only
be necessary to understand the behavior of the component rather than have intimate knowledge of the implementation and hardware details. Nor should a particular implementation inadvertently influence the design of the coordination algorithms. The controlled motor and motor group 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. The controlled motor abstraction is then used in a wheel abstraction and later a wheel locomotor model. The same paradigm applies at various levels of CLARAty.

Preliminary results showed that for one implementation of wheeled locomotor, around 90% of the implementation was reusable among FIDO, Rocky 8 and Rocky 7 (measured in lines of code which included comments). For the controlled motor, the reusable percentage ranged from 50%-70%. These statistics consider that all software drivers are non-reusable, even though they can be reused when boards share the same COTS chips.

6 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 high-fidelity simulation platforms. 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 these vehicles. We have presented a brief overview of the CLARAty architecture and some of the challenges in designing interoperable software that can run on varying physical rover platforms. 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.

7 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: