CLARAty: Challenges and Steps Toward Reusable Robotic Software

Issa A. Nesnas†; Reid Simmons‡, Daniel Gaines†, Clayton Kunz*, Antonio Diaz-Calderon†, Tara Estlin†, Richard Madison†, John Guineau†, Michael McHenry†, I-hsiang Shu†; & David Apfelbaum‡

†Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA

†(firstname.lastname@jpl.nasa.gov)

‡Carnegie Mellon, Pittsburgh, PA, USA

*Ames Research Center, Mountain View, CA, USA

Abstract: *We present in detail some of the challenges in developing reusable robotic software. We base that on our experience in developing the CLARAty robotics software, which is a generic object-oriented framework used for the integration of new algorithms in the areas of motion control, vision, manipulation, locomotion, navigation, localization, planning and execution. CLARAty was adapted to a number of heterogeneous robots with different mechanisms and hardware control architectures. In this paper, we also describe how we addressed some of these challenges in the development of the CLARAty software.*

1. Introduction

Within the NASA robotics community, and possibly within the research community, the majority of robotic software is designed and built from scratch for each new robot. To date, it may have been easier and more cost effective to do so. However, as robotic software gets more complicated and the time and effort to build reliable software increases, it becomes more important to develop reusable robotic software.

Over the past decade, NASA, through its Mars Technology Program, developed a series of rovers to mature new algorithms for planetary surface exploration. The Jet Propulsion Laboratory (JPL) developed the Rocky series, where its fourth generation culminated in the successful Sojourner rover that landed on Mars in 1997. Following that, JPL developed the FIDO series, which were precursors to the Mars Exploration Rovers (MER), Spirit and Opportunity, that landed in 2004.

NASA is interested in a reusable robotic framework to reduce the cost of integrating and testing new capabilities that are developed at various institutions. JPL started the research and development of reusable robotic software back in 1996 with the development of the Rocky 7 rover. The first generation reusable software was developed using a component architecture based on ControlShell (Volpe, 1997). The Rocky 8 software was, then, adapted

to this architecture. Due to limitations for supporting additional platforms and to find commonality in the development of robotic software among other centers, we developed the Coupled Layer Architecture for Robotic Autonomy (CLARAty, 2005). Started in 1999, CLARAty is the outcome of collaboration among JPL, Ames Research Center, and Carnegie Mellon (Volpe, 2001). In recent years, the University of Minnesota joined the team to develop the estimation framework.

Given the heterogeneity of the NASA research rovers, it was incumbent upon us to provide a framework that did not require the redesign of existing hardware. Additionally it was necessary to support legacy algorithms with significant investments.

While much of the context of this work focuses on robotic capabilities for planetary exploration, many of the challenges and approaches are more generally applicable.

2. Related Work

The development of general software architectures remains an active area of research in robotics (Coste-Maniere, 2000). Much of the effort focuses on hierarchical or layered architectures although there is disagreement over how to decompose the hierarchy. In the past, research focused on spatial or temporal hierarchies (Albus, 1991) and behavioral hierarchies

(Brooks, 1986). More recently, the focus has been on functional decomposition into different layers implemented with data structures and algorithms specialized for particular classes of functionality (Coste-Maniere, 2000). The most popular of such approaches is the three-tiered architecture (Bonasso, 1997) that features a declarative planning layer, a procedural real-time behavioral layer and an intermediate executive layer that mediates between the two. CLARAty decomposes robotic software into two layers: a decision layer and a functional layer. This approach is similar to the three-tiered architecture except that the planning and execution layers are combined in order to provide much tighter coordination between generation and execution of plans. A second difference between CLARAty and the three-tiered architecture is that CLARAty's robotic functionality can be accessed at different levels of abstraction. A somewhat different two-layered approach is CIRCA (Musliner, 1993) in which a planner/scheduler periodically creates and downloads policies to be executed in hard real time by a reactive control system. Unlike the hierarchy used in CLARAty's Functional Layer, the reactive layer in CIRCA has no internal structure, which makes it difficult to implement complex behaviors.

Other efforts in the robotics community aim at standardizing interfaces to robot hardware and among control processes. Probably the most visible effort is the Joint Architecture for Unmanned Systems (JAUS, 2005), which aims at providing standardized message passing interfaces for all of the military's unmanned vehicles. JAUS was initially developed by the Department of Defense to ensure interoperability among a family of Unmanned Ground Vehicles. Similar to CLARAty, JAUS defines interfaces that are independent of the integrated technology or the specific hardware platforms. While the goals of JAUS are similar to those of CLARAty, the approaches have significant differences. While the JAUS architecture uses a single-level message-set, CLARAty uses a multi-level abstraction model.

Another effort that provides abstractions for robotic devices is Player/Stage (Gerkey, 2003). Player/Stage is a device server that provides a flexible interface to a variety of sensors and actuators. It is based on a client/server model that uses socket-based communications. As a result, information exchange between components requires a serialization scheme, which can incur a significant cost for resource-constrained robots. Additionally, the current Player abstractions only address a limited set of capabilities primarily geared towards controlling commercial-off-the-shelf robots with simple mobility mechanisms.

The Foundation for Intelligent Physical Agents (FIPA) is a similar effort in the world of multi-agent systems. Unlike CLARAty, both FIPA and Player/Stage focus on the form of the interfaces and less on their content. They are also aimed mainly at the lower-level control aspects, whereas CLARAty tries to address a more complex functional hierarchy.

More recently, there have been several other related efforts driven by similar needs. We will only list two: The OROCOS project (OROCOS, 2005), which provides both hard real-time services and class libraries for robotic applications; and the OSCAR project (OSCAR, 2005), which uses a similar object-oriented decomposition to that of CLARAty for analysis, control, and simulation of manipulators.

3. Challenges

Developing reusable robotic software is difficult primarily due to the variability in robotic platforms. Initially, one may assume that by concretely defining the content and rate of information flow among the various subsystems, one establishes a plug-and-play robotic architecture. While defining the information and its flow is necessary, it is not sufficient. The content and pathways of the information flow change with various device and system configurations, as well as with different application programs. Hence, the flow of information among sub-systems has to be both flexible and efficient. To reuse software components across a wide range of systems, it is also important that components of a robotic system make no assumptions about their operational platforms. Therefore, it is necessary to share configuration, kinematic, and dynamic information among components.

This section presents four major challenges that stem from trying to: (i) control heterogeneous robots, (ii) integrate and interoperate new capabilities, (iii) adjust access levels, and (iv) implement a generic framework. In the next section, we will present some of the approaches that we used in CLARAty to overcome these challenges. The list of challenges below is not intended to be exhaustive, but rather characteristic of the key challenges that we faced in standardizing the development of robotic software.

3.1. Control Heterogeneous Robots
Because there are no standard robotic platforms, any reusable framework must be sufficiently flexible to address the variations in robots. Robotic systems present challenges due to differences in their physical capabilities, sensor configuration, and hardware control architectures.

The first challenge comes from physical variability. Consider the example of mobile rovers. Within this class of rovers, there are wheeled rovers, legged rovers, and rovers that are a hybrid of the two. Even within the wheeled rover subclass, platforms have different mobility mechanisms and wheel configurations. Some have four wheels while others have six or eight wheels. Some have all-wheels steering while others have only
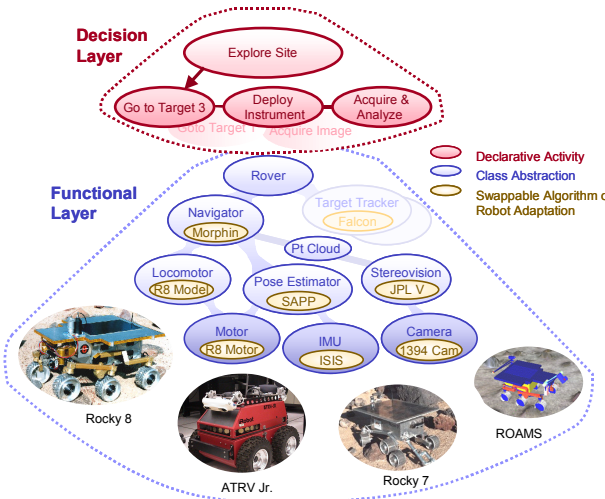
Figure 1: The CLARAty Architecture

front-wheels steering. Figure 1 shows examples of various robots.

Because of this physical variability, these robots possess different capabilities. Fully-steerable (omni-directional) rovers can move laterally (crab) while partially-steerable (car-like) rovers have to use parallel parking maneuvers to obtain the same result. For software to interoperate across such platforms, it has to provide a generic interface that handles these constraints.

When crafting a generalized interface, it is often the case that neither the union of all possible capabilities nor the intersection of such capabilities (least common denominator) is satisfactory. The solution often lies somewhere in between. In some cases, it is necessary to split the interface into two distinct units and lose the ability to interoperate between the two. This occurs when it is necessary to highlight the differences between platforms rather than their commonality. Trying to find the single unified interface can sometimes lead to undesirable over generalizations.

The second challenge comes from differences in sensor configurations. One situation is where different sensors produce similar information, but have different physical constraints. For instance, consider sensors that generate terrain data that is represented as three-dimensional point clouds. To generate this data, one can either use a lidar or a stereo camera pair. While both devices eventually generate point clouds, these two devices operate with different constraints and have different qualities. A lidar requires a longer time to scan a scene but less time to generate the depth information, while the opposite is true for stereo. These behavioral differences generate constraints on the operation of the vehicle. Because of these variations in sensors, it is necessary for algorithms to interface to an appropriate abstraction of that sensor rather than to the actual sensor driver. If a navigation algorithm that uses this data to find obstacles was interfaced to stereo cameras as opposed to point clouds, then it will not be possible to use this algorithm on rovers that use a lidar sensor in lieu of stereo cameras.

Another situation is where similar data can be produced by either a single sensor or a suite of sensors. For instance, consider inertial motion sensing. Some robots may use individual gyroscopes and accelerometers in a unique configuration to measure the rover's ego motions. Others may use an integrated Inertial Measurement Unit. In the first case, the hardware/software framework must ensure the synchronized acquisition and processing of these raw measurements, while in the second case, the interface to the IMU provides such capability.

The third challenge comes from differences in hardware control architectures. On one end of the spectrum, there are robots that use a centralized processor to servo the motors, generate coordination trajectories, and run the application software. Such systems often have signals mapped to memory registers in a centralized processor making the development of software relatively easy. However, they lack in modularity and are hard to extend. On the other end of the spectrum, there are systems that move as much of their controls as possible to firmware in embedded distributed nodes in order to reduce the load and real-time requirements on the central processor. Other systems fall somewhere within this spectrum. While each approach has its pros and cons, a general framework must handle these differences in hardware control architecture that has significant impact on information flow.

In addition to these challenges that are found in both custom designed robots as well as commercial-off-the-shelf robots, components that comprise a robotic system continue to change. Image acquisition subsystems changed from analog cameras with centralized framegrabbers to distributed digital cameras connected to FireWire or USB buses. Despite this, reusable robotic software must be sufficiently flexible to support such variations or rapidly adapt the software to handle them.

3.2. Integrate and Interoperate New Capabilities

The integration of algorithms is perhaps one of the most challenging elements in developing reusable robotic software. The challenges stem from trying to integrate algorithms that use different representations of information and different architectures.

The first challenge is in the multiple ways to represent similar information. Consider, for instance, general transformations that connect one coordinate frame to another. Two ways for representing the orientation portion of these transformations are rotation matrices and quaternions. These representations have different characteristics in terms of efficiency and ease of use/understanding. Conversion between them is both inefficient and error prone. This is especially true when dealing with their covariances. In addition,

transformations cannot be defined in isolation. They require a context that defines the relationships between frames and whether those frames represent fixed or articulated connections. Without agreement on these representations, algorithms will be required to deal with these conversions in an *ad hoc* manner, leading to loosely integrated and inefficient software.

The second challenge stems from architectural mismatches. One issue is with components that integrate orthogonal functionality into a single modular unit. This introduces artificial coupling of functionalities driven by a specific implementation. While such coupling may have some locally optimal performance, this often comes on the expense of global optimality.

Another issue is with limitations of architectural frameworks. Consider, for instance, a framework that does not time-stamp measurements collected from various devices. Now consider an algorithm that collects data asynchronously and requires time-stamped measurements. If the underlying framework does not support time-stamped measurements, this results in an architectural mismatch. Similar situations occur when an algorithm requires high bandwidth information that may not be available for certain platforms. Diagnostic and health monitoring software often requires information about all aspects of the system at all times, which in many cases, may be limited or not possible.

The interoperation of algorithms necessitates components that produce a similar output even though they may use different underlying technologies. The challenge is to provide a framework in which a developer can work on an individual technology component and see how it interacts with a complete robotic system, without having to understand the entire system.

3.3. Adjust Access and Control Levels

In any complex system, it is important to be able to access and independently test each subsystem. Characterizing the performance of individual subsystems requires a modular architecture that provides access at various levels of the architecture. Also, interfacing with other systems requires adaptations at different levels of granularity. This section covers both back-end and front-end access. Back-end access is what gets adapted to hardware or simulation. Front-end access is what a client application uses to control the software.

With respect to back-end access, consider the FIDO rover, which uses a central processor for the control and coordination of its motors. The software framework must provide functionality for the servo control and trajectory generation for all motors. In this case, the interface to hardware occurs at the low-level of digital and analog I/O. However, in systems such as Rocky 8 or Rocky 7, which use micro-controllers for servo control and trajectory generation, the interface to hardware occurs through communication with motor controllers.

Providing multi-level access also benefits interfacing the control software with simulation. Some simulations may not have the level of fidelity to simulate real hardware. In such cases, a higher-level interface would be necessary and appropriate. There are other cases where a higher-level interface is desirable to explore a larger set of scenarios without having to go through smaller steps of motion simulation. For example, an interface between the control and simulation software at the locomotion level bypasses the lower level control and simulation of actual wheel motions.

With respect to front-end access, client applications may need to access the system at different levels at different times. For example, if the robotic arm has on-board autonomy for path planning, then one uses the high-level interface to define goal locations. However, in other situations where the arm has to be tele-operated, one needs to interface to lower-level motor velocities.

3.4. Implement a Generic Software Framework

Another major challenge stems from the inherent complexity and multi-disciplinary nature of the robotics domain. Developing robotic capabilities for real systems is quite hard, but doing so with an overarching objective of supporting new platforms and algorithms that are not known *a priori* is a real challenge. This process requires developers with both a depth of knowledge in robotics and breadth of experience and skills in the field.

Developing a cross-cutting generic framework requires continuous refactoring of common elements across multiple disciplines. There are shared capabilities among the vision, mobility, and manipulation domains. They all require coordinate transformations, math libraries, and information about the mechanisms they control. Similarly, the science analysis and vision domains share abundant image processing infrastructure.

To keep the complexity of systems manageable, and to simplify the testing and maintenance of the various packages, it is important to reduce code duplication as much as possible across domains. This raises the question of when it is appropriate to encapsulate an algorithm vs. to refactor it using a common software framework. The decision is often influenced by non-technical factors involving the nature of the technology, the expertise necessary to re-implement the algorithm, the return of investment, and the long-term plan to support the algorithm as part of a common framework. Because any reusable robotic system is doomed to become enormous, it is strongly desirable to make the code repository complementary rather than duplicative.

To support the integration of multiple algorithms and to support the adaptations of the framework to various robot platforms, it is necessary to have development tools and processes that support modularity. Without the ability to check out and build parts of the generic robotic repository, it becomes too unwieldy to use. The

repository tools will eventually need to be integrated with the build system in order to dynamically check-out and configure the system using different implementations of a given functionality. For instance, consider rover navigation that can use one of three algorithms for estimating the rover pose: a wheel odometry pose estimator, a visual odometry pose estimator, or multi-sensor pose estimator. Depending on the desired configuration, the software check-out and build will be different for each of the three configurations. Automated tools are needed to provide such flexibility.

In addition, there are many challenges in software engineering that any generic framework for robotics will have to address. No matter what the approach used in the design, issues related to the flexibility, scalability of the approach, simplicity but not simplistic, extendibility, and long-term maintainability can only be judged over time. The challenge is to find the delicate balance among the above.

3.5. Address other Challenges

Numerous other challenges remain in the development of a unified and reusable robotic framework, but it will suffice, here, to point to a few more. Some of these challenges include: dealing with system states especially ones that reside in hardware controllers where there is a cost associated retrieving state information; logging of information at all levels; dealing with measurement uncertainties; dealing with differences in data flow models among platforms; dealing with multiple clients; supporting real-time operations; addressing abstract time for real and simulated platforms; and addressing distributed computing nodes. There are several additional social factors such as getting user buy-in, managing contributions from a distributed developer base, capturing feedback from the user community, and providing documentation, training and support.

4. CLARAty

CLARAty is a reusable robotic software framework to enable the integration of new capabilities onto various platforms. We designed CLARAty to address the above challenges in software interoperability for rovers and manipulation platforms. CLARAty defines standard interfaces at different levels of abstractions for various devices and robotic algorithms. It also provides candidate implementations for each algorithm as a starting point, though many algorithms were contributed through a competed program by robotic developers at universities and NASA centers. In addition to interfaces and algorithms, CLARAty also provides adaptations of its device abstractions to custom and standard hardware and robotic platforms. The CLARAty code base is designed with a modular structure to enable users to check out and work with only the parts of the software that meets their needs. The majority of the software is developed using object-oriented C++.

In the following subsections, we will address the aforementioned challenges in an order that facilitates the description of some elements of CLARAty.

4.1. Approach

Because it is not realistic to expect a standard robotic platform any time soon, it becomes necessary to develop a software framework that would deal with the variability outlined in the previous section. To do so, we analyzed in detail several existing robotic architectures and legacy implementations of several NASA robots, including Rocky 7, Rocky 8, FIDO, K9 and Dexter. We also investigated the interactions between declarative model-based reasoning and these architectures.

To meet the flexibility requirements for integrating different technologies, we developed CLARAty as a two-layer architecture with the top decision layer and a bottom functional layer. The decision layer uses a declarative model-based approach to define activities. The input to this layer does not *a priori* specify the order of execution of activities. Rather, activities are described with explicit system and mission constraints and a search engine orders these activities at runtime to provide a feasible plan. The plan is then executed using an engine that is tightly integrated with the planner. The functional layer, on the other hand, uses an object-oriented procedural approach where the sequence of execution is defined *a priori* and bounded by the software implementation. Hence, the system does not have to search for a feasible plan before execution.

We architected the functional layer to use a multi-level abstraction model with polymorphic interfaces to address the variability of robotic systems. At the mission level, a robot can plan and execute a number of activities in different order. At this level, a declarative model dominates. However, the choices for actions become limited and time-constrained as you go down the hierarchy. At these levels, a procedural model dominates. Most robotic systems use both models (Coste-Maniere, 2000). Where one layer ends and another begins remains an active area of research. Current practice has drawn the line between the two models at a high level, however, in CLARAty, the decision layer can access the functional layer at different levels.

To address challenges in software implementation, we leverage many well-known techniques developed by the software community, including object-oriented architecture, design patterns, generic programming and component-based architecture (Gamma, 95) (Garlan, 1996). Our experience shows that an object-oriented framework provides the necessary levels of abstraction to deal with the variability among platforms and algorithms. It also provides extendible interfaces, strong type checking, polymorphic behavior, and data encapsulation, which are all necessary elements for the robust development of complex robotic systems. Most

component interactions use method calls on class abstractions with only a few that use the more elaborate component-connector style interface. The latter is primarily used when distributing computation across nodes is necessary (e.g. the interface between the decision and functional layers), which requires serialization and de-serialization of commands and information. Component-based architectures such as MDS (Dvorak, 1999) and ControlShell (Pardo-Castellote, 1998) require additional frameworks for the explicit ordering and have coarser granularity for parallel execution of activities. CLARAty, on the other hand uses the multi-threading model of its operating system to provide finer resolution on the scheduling and pre-emption of activities. That requires though a multi-thread safe implementation of these algorithms.

4.2. A Multi-level Abstraction Architecture

The system is designed with abstractions at various levels from the low-level device abstractions to high-level functional abstractions. At the lowest levels are device abstractions that get adapted to various platforms. These include analog and digital I/O, motor, IMU, camera, and spectrometer abstractions. At higher levels are abstractions that integrate various lower-level abstractions. Examples of these abstractions include locomotor, manipulator, pose estimator, navigator, and rover. Higher-level abstractions provide interfaces for different robotic algorithms. A more detailed description of the architecture and class abstractions can be found in Nesnas (Nesnas, 2003).

In addressing architecture mismatches, there is often a fundamental tension between the desire to separate abstractions for conceptually distinct parts of the system and the reality of the coupling between hardware and software components. Consider, for instance, a camera that is powered by a power distribution subsystem. The camera device and the power subsystem have distinct functionality, and we would like to keep the implementation of their interfaces independent and modular. However, at some point, a camera will have to be switched on/off. So the question arises: should the user ask the power system to turn the camera on, or should the user ask the camera to switch itself on? In the first case, the user has to know about the power system, and in the second case, the camera has to know about the power system. Neither case is ideal. For someone who cares only about images, the power system is a nuisance; for system designer, the dependency between the camera and the power system leads to a break in modularity. We address this type of problem by using light-weight function objects (functors). An abstract power functor provides an interface to turn a device on/off and to measure its voltage and current draw. The power distribution system then creates these objects on request and gives them to devices as they are built. Some part of the initialization code, therefore, needs to know the coupling between the power distribution subsystem and the camera. However, using this approach, cameras are not aware of the underlying implementation of power switching, and users can now ask the camera directly to turn itself on or to report on its current draw.

To operate the software on real and simulated platforms and to support "what if" planning scenarios, we separate mechanism models from their controls. To address the variability of different mechanisms, we use flexible abstractions that capture the model characteristics for use by various applications. This modeling captures geometric information in order to support collision prediction and detection for safe robot operations. Typical robot applications require forward and inverse kinematics algorithms. We will provide generic solvers for the kinematics and inverse dynamics for the generic model framework. Because some applications require high-speed robot motions with tight control loops, we support the overriding of the generic solvers with more efficient mechanism-specific implementations. We define a set of abstractions to also describe the interactions and contacts of the mechanism with its environment. For more details on the mechanism modeling in CLARAty, please refer to Diaz-Calderon (Diaz-Calderon, 2005).

One of the main features of CLARAty is its ability to interoperate robotic algorithms. There are many challenges that make this difficult, including the problem of making algorithms themselves generic in the first place. At first, it may seem easy to provide a common API to a collection of, say, stereo algorithms: the primary interface takes a pair of images with their camera models and produces a disparity map. This seemingly abstract interface fails with the first step of most stereo algorithms, when images are rectified to remove lens distortion and ensure epipolar alignment. This is because legacy implementations of stereo algorithms typically perform rectification internally, and the algorithm for producing the rectification depends on the underlying implementation of the camera model. There are several possibilities to make the stereo vision API truly generic: (i) pass in only rectified and aligned images to the stereo algorithm without needing to pass in camera models, or (ii) pass in images with their corresponding camera models, however, have the camera models implement rectification (both to remove lens distortion and epipolar align images). Each of these implementations has its own drawbacks; the usual tradeoff is between simplicity and performance. Epipolar alignment is primarily useful for stereo, so making it a requirement on the camera model class is somewhat awkward and a burden to implementers of new models, such as push broom camera models. On the other hand, requiring the user to rectify images before handing them to a stereo algorithm is also something of a burden, particularly if the user must take extra steps to keep the rectification efficient, for example when batch processing several image pairs that all have the same epipolar relationship. In this case, we prefer the solution that keeps the interface generic. This means that more work is required when integrating

legacy algorithms into the system, and shows that the most abstract interface is not necessarily going to be the simplest. However, with a truly abstract interface to stereo algorithms, a user will be able to mix and match camera models and stereo implementations to find the best combination of components for a particular application. This flexibility, more than makes up for the additional complexity that the user must address.

Autonomous navigation, which provides obstacle avoidance capabilities for mobile robots, uses many of the lower level capabilities, such as vehicle locomotor, point cloud sources, local and global path planners, and pose estimators. First, a generic interface was designed that allows higher levels to invoke the navigation functionality in the same way, regardless of what algorithm is actually being used or which rover is being controlled. This "navigation" interface basically indicates goal points (or, more generally, goal regions) that the robot must reach. Navigation algorithms are then adapted to this framework to accept input from the CLARAty point cloud source and command the rover using the vehicle locomotor, a generic interface to a wide range of supported rovers. More fundamentally, however, navigation algorithms that were adapted to CLARAty all had to be extended to plan generically for different rovers. For instance, the algorithms all need to know the maximum steering angles to determine how tight turns can be made and the size of the rover to determine what distance between obstacles constitutes a safe passage. This was accomplished with the mechanism model described above. In addition, to support the Morphin algorithm (Urmson, 2003), the mechanism model class can perform a kinematic simulation of the rover. This enables the algorithm to integrate costs along the rover's path without having to know explicitly how the rover moves. In current work, Carnegie Mellon is developing navigation algorithms that take vehicle dynamics into account, and we expect to extend the mechanism model to support dynamical simulation, as well.

On the various access and integration levels, algorithms can be integrated into CLARAty in different ways. Some algorithms can be encapsulated behind the generic CLARAty APIs, while others can be refactored to leverage CLARAty's data structures and generic classes that we believe may be useful for many different algorithms. Refactoring algorithms enables more efficient and consistent representation of the internals of an algorithm. For instance, the GESTALT (Goldberg, 2002) algorithm that was flown on MER rovers was encapsulated into CLARAty while the Morphin algorithm was refactored. Currently, we are refactoring the Drivemaps algorithm. The goal is to determine how much reuse can be made from algorithms that have fundamentally different approaches to the same problem. While complete reuse of the classes is unlikely, we have found that splitting the algorithms into terrain analysis

and action selection components seems to be common amongst the algorithms that we have investigated to date.

4.3. Empirical Results

We have developed autonomous end-to-end rover capabilities such as autonomously placing an instrument on a target selected from 10 meters away. Such capability integrates visual tracking of the designated target using multiple rover mounted cameras while navigating to the target location; assessing the safety of the target region; properly positioning the rover relative to the target for instrument deployment; deploying and placing the robotic arm that carries the science instrument on the target; acquiring the scientific data and simulating a downlink to Earth.

We have deployed and extensively tested CLARAty on half a dozen robotic platforms. Figure 1 shows a subset of these platforms, which include the custom Rocky 8, FIDO, Rocky 7, and K9 rovers, as well as the ATRV Jr. COTS platform. These platforms have different mobility mechanisms and wheel configurations as well as different sensor suites, manipulators, end effectors, processors, motion control architectures and operating systems. In addition to these real-platform adaptations, we have also adapted CLARAty to operate with the high-fidelity ROAMS rover and terrain simulator (Jain, 2004).

A large number of complex algorithms have been integrated into CLARAty and deployed on the above platforms. For autonomous navigation, we have integrated the GESTALT algorithm that is driving the MER rovers today on the Martian surface (Goldberg, 2002), the Morphin algorithm that GESTALT was based on (Urmson, 2003), and the Drivemaps algorithm (Huntsberger, 2001). In each case, the original implementation had to be modified and generalized in relatively minor ways to fit the CLARAty framework. For rover pose estimation, we have adapted five algorithms including the Sojourner algorithm (Mishkin, 1998), the MER pose estimator algorithm, and a new algorithm that integrates all rover sensing modalities (Roumeliotis, 2002). These algorithms all require data from different sensors, including wheel encoders, gyroscopes, IMUs, sun sensors and stereo cameras. We also integrated three stereo vision algorithms, several visual target trackers, visual odometry, sensor-based manipulation, path planning, science analysis, and automated planning and scheduling. Many of these algorithms have been tested on multiple platforms and as part of end-to-end capabilities.

5. Conclusion

Developing reusable robotic software presents many challenges. These challenges stem from variability in robotic mechanisms, sensor configurations, and hardware control architectures. They also stem from integrating new capabilities that use different representations of information or that have architectural mismatches with the reusable framework. We found that multi-level

abstraction models, object-oriented methodologies and design patterns go a long way to address the extensive variability that is encountered in today's robotic platforms. We have learned that over-generalizing interfaces makes them harder to understand and use. There is a delicate balance between flexibility and simplicity. Performance cannot be compromised for the sake of flexibility and least common denominator solutions are often unacceptable. It is necessary to have flexible development environments, tools, solid regression testing. There is also no substitute for well-documented products and development processes. It would be highly desirable to standardize robotic hardware but that may not be feasible today.

6. Acknowledgments

7. References

Albus, J. (1991). Outline for a Theory of Intelligence, IEEE Transactions on Systems, Man and Cybernetics, 21:3, pp. 473-509, 1991.

Bonasso, R.P.; Firby; Gat, G.; Kortenkamp, D.; Miller, D.; & Slack, M. (1997). Experiences with an Architecture for Intelligent, Reactive Agents, Journal of Experimental and Theoretical Artificial Intelligence, 9:2, 1997.

Brooks, R. (1986). A Robust Layered Control System for a Mobile Robot. IEEE Journal of Robotics and Automation, RA-2:1, 1986.

Coste-Maniere & E.; Simmons, R. (2000). Architecture, the Backbone of Robotic Systems. IEEE Conference on Robotics and Automation, San Francisco CA.

CLARAty (2005), http://claraty.jpl.nasa.gov

Diaz-Calderon, A.; Nesnas, I.; Kim, W.S.; & Nayar, H. (2005). Towards a Unified Representation of Mechanisms for Robotic Control Software. Submitted to the Int'l Journal of Advanced Robotic Systems.

Dvorak, D; Rasmussen, R.; Reeves, G.; & Sacks, A. (1999). Software architecture themes in JPL's Mission Data System, Proc. of the AIAA Guidance, Navigation, and Control Conference, Portland, OR.

FIPA (2005). Foundation for Intelligent Physical Agents, http://www.fipa.org/.

Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. (1995) Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series.

Garlan, D. & Shaw, M. (1996). Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall.

Gerkey, B.; Vaughan, B.; & Howard, A. (2003). The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems, International Conference on Advanced Robotics, pages 317-323, Portugal.

Goldberg, S.; Maimone, M.; & Matthies, L. (2002). Stereo Vision and Rover Navigation Software for Planetary Exploration, Proceedings of the IEEE Aerospace Conference, pp 2025-2036.

Huntsberger, T.; Aghazarian, H.; et.al. (2002). Rover Autonomy for Long Range Navigation and Science Data Acquisition on Planetary Surfaces, IEEE International Conference on Robotics and Automation, Washington, DC, pp. 3161-3168

Jain, A.; Balaram, J.; Cameron, J.; Guineau, J.; Lim, C.; Pomerantz, M.; & Sohl, G. (2004). Recent Developments in the ROAMS Planetary Rover Simulation Environment. IEEE Aerospace Conference, Montana, 2004.

JAUS (2005). Joint Architecture for Unmanned Systems Reference Architecture, Version 3.0, http://www.jauswg.org/.

Kapoor, C; & Tesar, D. (1998). A Reusable Operational Software Architecture for Advanced Robotics, CSIM-IFToMM Symposium on theory and Practice of Robots and Manipulators, Paris, France.

Mishkin, A.; Morrison, J.; Nguyen, T., Stone H.; Cooper, B.; & Wilcox, B. (1998) Experiences with Operations and Autonomy of the Mars Pathfinder Microrover, IEEE Aerospace Conference, Colorado.

Musliner, D.; Durfee, E. & Shin, K. (1993) IRCA: A Cooperative Intelligent Real-Time Control Architecture. IEEE Transactions on Systems, Man and Cybernetics, 23:6, 1993.

Nesnas, I.A., Wright, A., Bajracharya, M., Simmons, R., Estlin, T., Kim, W.S. (2003) CLARAty: An Architecture for Reusable Robotic Software, SPIE Aerosense Conference, Florida.

OROCOS (2005) http://www.orocos.org/

Pardo-Castellote, G.; Schneider, S.; Chen, V.; and Wang, H. (1998) ControlShell: A software architecture for complex electromechanical systems, Int'l Journal of Robotics Research, 17(4).

Roumeliostis, S.; Johnson, A.; & Montgomery, J. (2002) Augmenting Inertial Navigation with Image-Based Motion Estimation. IEEE International Conference on Robotics and Automation, Washington D.C.

Simmons, R.; & Krotkov, E. (1995). Experience with Rover Navigation for Lunar-Like Terrains, Conference on Intelligent Robots and Systems.

Volpe, R.; Nesnas, I.A.; Estlin, T.; Mutz, D.; Petras, R.; & Das, H. (2001) The CLARAty Architecture for Robotic Autonomy. IEEE Aerospace Conference, Montana,

Volpe, R.; Balaram, J.; Ohm, T.; & Ivlev, R. (1997). Rocky 7: A Next Generation Mars Rover Prototype. Journal of Advanced Robotics, 11(4).