The CLARAty Project: Coping with Hardware and Software Heterogeneity

Issa A.D. Nesnas

Jet Propulsion Laboratory, California Institute of Technology
nesnas@jpl.nasa.gov

1 Introduction

Developing reusable robotic software is particularly difficult because there is no universally agreed upon definition of what a robot is. Over the past half-century, robots took many forms. Some were inspired by their biological counterparts such as humanoids, dogs, snakes, and spiders. Others were designed for particular domains such as manufacturing, medical, service, military, or space applications. They took the form of arms, wheeled robots, legged robots, hoppers, blimps, underwater vehicles, sub-surface diggers, and even reconfigurable robots.

While it is neither practical nor possible to address heterogeneity across all types of robots, there are some common themes that recur in robotic software. In this chapter, we will summarize the challenges of developing reusable software for heterogeneous robots and present some principles for coping with this variability.

We arrived at these principles by doing a variability and commonality analysis and building an application framework for a class of heterogeneous robots. Our goal is to improve the interoperability of advanced robotic algorithms through the reuse of the software that implements these algorithms. This framework is called *CLARAty* , which stands for *$\underline{C}$oupled-$\underline{L}$ayer $\underline{A}$rchitecture for $\underline{R}$obotic $\underline{A}$u$\underline{t}$onom$\underline{y}$* [cla06]. It is a joint collaboration among the Jet Propulsion Laboratory, NASA Ames Research Center, Carnegie Mellon, and the University of Minnesota. CLARAty is the framework for integrating, maturing, and validating new technologies developed by institutions and universities under NASA's Mars Technology Program.

We have built this framework by generalizing legacy software of multiple robots developed over a decade. We considered different architectural styles to improve software reuse and to ensure scalability of the framework. Later, we extended our framework to a larger set of platforms expanding both the scope and capability, which required a more general domain analysis. This iterative process of design, implementation, deployment, testing, and capturing of lessons learned from real systems is critical to reaching reusable and stable robotic software.

Finding the right level of software generalization depends on the scope of their applicability and the life cycle of the robots. Writing application software against generalized and stable components allows upgrading robot hardware without having to rewrite the application software. This has been true over the life cycle of several of our robots.

Efforts to build reusable robotic software and frameworks date back several decades. These efforts have primarily been driven by a pragmatic need to structure the development of software to simplify the building of larger systems. These included developing reusable robotic software libraries [HP86] and developing application frameworks [AML87] [PCSCW98] [SVK97].

Despite earlier efforts, integrating robotic capabilities on different platforms remained quite difficult. The desire for interoperability of robotic software continued, which led to renewed efforts [jau06] [NWB03] [Hat03] [ACF98] [Alb00] [VG06] [KT98], to name a few. While many techniques have been proposed over the years, the primary challenge remains in the poor scalability and lack of flexibility to handle the heterogeneity of robotic software and hardware.

2 Challenges

Two paradigms have emerged for reusing software in robotics: (a) a component-based approach where the components are concrete reusable elements and (b) an object-oriented approach where the reusable components are generic abstractions and interfaces that get adapted to a particular context. In the first model, components are concrete building blocks that achieve specific functionalities. They are then connected to each other statically or dynamically using architecture description languages (ADLs). In the second model, components are separated into generic base classes, which define the generic interfaces and interactions with other classes, and specialized classes that adapt the generic functionality to a given platform. Classes are connected through interfaces that either statically or dynamically bind to other classes. In this Chapter, we will primarily focus on the second model. We will present the challenges in developing reusable robotic software and present techniques for coping with the heterogeneity of hardware and software.

The rest of the chapter is structured into two main parts. The first part addresses the challenges of software variability that stems from (a) software complexity, (b) algorithm integration, (c) architectural mismatches, (d) software efficiency, and (e) multiple operating systems and tools. The second part addresses the challenges of hardware variability that stems from (a) hardware architectures, (b) hardware components, (c) sensor configurations, and (d) different mechanisms.

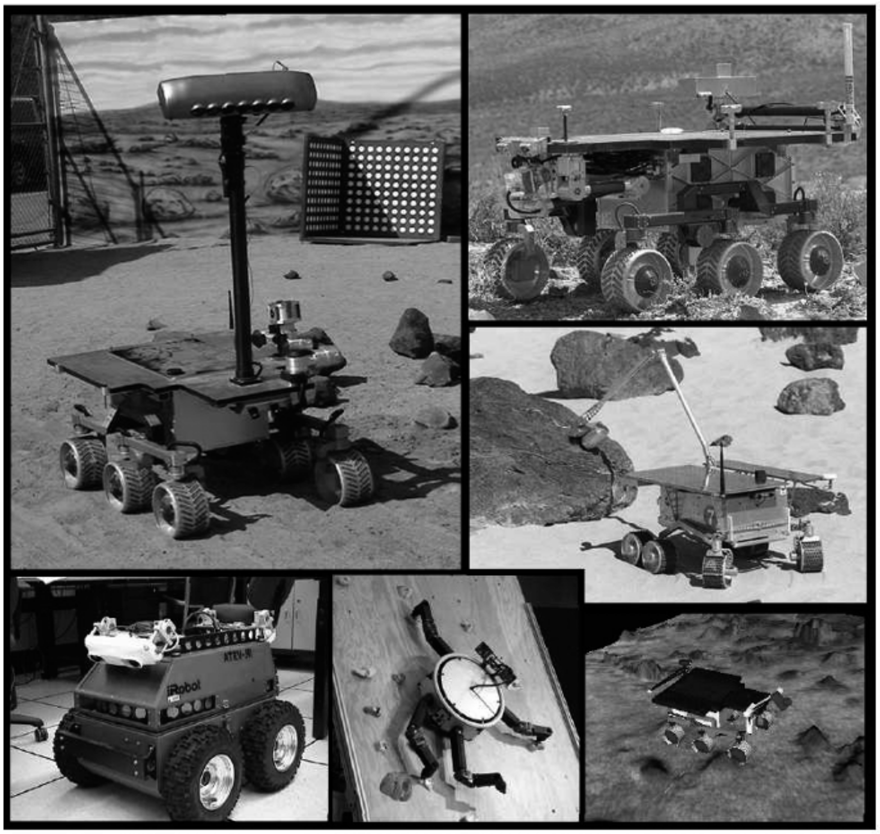

Fig. 1. From top left and clockwise: *Rocky 8*, FIDO, *Rocky 7*, ROAMS simulation, Lemur II, and ATRV Jr. robots

Each part briefly describes the challenge and proposes one or more solutions to cope with the challenge. We illustrate these with examples from our systems. The list of challenges is not intended to be exhaustive, but rather characteristic of the key areas that are common in standardizing the development of robotic software. A more comprehensive discussion of these challenges can be found in [NSG06]

We will conclude with a brief description of the software capabilities that we integrated into the *CLARAty* framework using these techniques. We have successfully interoperated these capabilities on most of the systems shown in Fig. 1, which include the *Rocky 7*, *Rocky 8*, FIDO, K9, Dexter, and ATRV Jr. rovers as well as the ROAMS simulation.

In this chapter, we will use the term *component* to refer to the software that implements a concrete capability. We will use *generic abstraction* (or abstract class) to define a capability and an interface devoid of its implementation. We will use the term *module* to refer to a collection of abstractions (i.e. classes) that have high cohesion and *package* to define a collection of modules that represent a domain.

3 Software Variability

3.1 Software Complexity

Challenge: How to Decompose a Robotic System

Developing robotic software is already complex because of its multi-disciplinary nature, but doing so with an objective of supporting future platforms and algorithms is a real challenge. This process requires deep knowledge, broad experience and an anticipation of future capabilities and platforms.

Solution: Decompose to highlight stable behaviors and not run-time implementations

Some system decompositions highlight the runtime model of the system, while others highlight the abstract behavior of the components hiding the runtime implementation. It is very likely that future platforms will have different hardware architectures, which would require different runtime models. Therefore, it necessary to encapsulate runtime models and highlight abstract behaviors of components, which are more stable across applications.

Example: Image acquisition

To illustrate this point, consider the example of an imaging system. The primary function of such a system is to acquire images. How the imaging system acquires the images depends, largely, on the underlying hardware. In some systems, an analog camera is connected to a frame grabber that is mounted in a backplane bus (e.g. cPCI). In other systems, a digital camera is used to transmit images over a fast serial interface directly to the host's memory. In either case, the primary function of the imaging system remains the same, i.e. to acquire images or a video stream. We can represent such a system by a camera abstraction that publishes a uniform interface but hides the details of its implementation and the runtime models that accomplish the image acquisition.

Challenge: How to Organize the Software

Complex applications require a large amount of software to be managed, integrated and deployed. The primary challenge in decomposing the system is to define where to draw the lines. This largely depends on what elements of the software are targeted for reuse by future applications.

Solution 1: Decompose into small modules rather than large packages

It is natural to think of decomposing the system into packages that reflect the robotic domains. However, such decomposition would overlook common themes across domains. Working at the granularity of packages would also incur a larger overhead especially when few software capabilities are needed by an application. The guiding principle here is to ensure that simple functionalities are light-weight and easy to implement, while complicated functionalities can be more complex.

Therefore, it is preferable to decompose robotic software into smaller modules as opposed to larger packages. Each module provides primarily a single capability but contains a collection of software with high cohesion and fewer interactions with other modules. A system will then be composed of a number of inter-dependent modules. A module usually contains a set of abstractions that are closely related to one another and that are managed as a group. A module defines the smallest deployable software collection. It is managed as a unit for repository access, building, and testing. Packages are then a loose grouping of modules where multiple packages share common modules. These are primarily used to simplify the description of an overall system.

Example: Vision package

Consider a vision package that contains the following modules: camera models, stereo processor, visual odometer, structure-from-motion, visual tracker, object finder and template matcher. An application that only requires a stereo-vision algorithm would only need to use three modules: stereo processor, template matcher and camera models. There would be no need to checkout and build the remaining modules. Without the ability to work at the granularity of modules, one would have to carry the weight of irrelevant modules in a package. This problem gets compounded for multiple domains such as vision-guided manipulation.

Solution 2: Use explicit inter-module dependencies

Communicating using strongly typed messages ensures properly matched interfaces for information flow across modules. It also enables the tracking of interface changes at compile-time rather than at run-time. Modules that are not interdependent, on the other hand, would not share a common infrastructure and would end up with implicit dependencies on data format for information exchange. Both the format and context of these data packets would need to be tracked and verified to ensure compatibility between senders and receivers. While a loosely coupled system may be easier to initially build and interface with, its software would be harder to extend and maintain over its life cycle.

Using explicit inter-module dependencies also enables the automation of inter-module dependencies. This is important as the number of modules grow in the system. Without the ability to check out and build parts of a generic robotic repository, it becomes too complex and unwieldy to use.

Example: Checking out modules

Because robotics is multi-disciplinary, it is not unreasonable to expect hundreds of modules especially as each capability may have multiple implementations from different institutions. Today, the *CLARAty* software repository contains over 400 modules, which includes hardware adaptations to half a dozen platforms. Because any given application only exercises parts of this reusable software repository, we use an automated process for tracking module dependencies to hide the complexity of managing these dependencies. For example, checking out and building the robotic manipulation software requires modules for arm control, motor, trajectory generator, and mechanism model. It would not be necessary to include other modules.

In a second example, a rover navigation component can use one of three components for estimating the rover pose: wheel-odometry, visual-odometry, or an inertial-based pose estimator. Depending on the desired capability, the software checkout and build will be different for each configuration.

Solution 3: Define a common vocabulary

Decomposing the system into modules and further into components defines a common vocabulary that describes the entities and their functionality. There are two types of modules: generic and specific. Generic modules declare abstract interfaces that describe the language of inter-module interactions. The interactions of these abstractions define the generic framework. Specific modules adapt the abstractions and hence the framework to various algorithms and platforms.

Example: Locomotor and stereovision

A *locomotor* module describes the generic abstractions and interfaces for mobile robots. This module with its limited interface is general enough to support any wheeled, legged or hybrid robot. The more specific *wheel locomotor* module provides a richer interface for all wheeled vehicles (Fig. 15).

An example of generic and specific modules relates to the stereovision capability. A generic *stereovision* module contains the data structures and interfaces that define the stereovision capability. The specific stereovision modules contain different stereovision algorithms that implement that interface. Each stereovision implementation resides in its own module.

Solution 4: Avoid unnecessary code duplication and overgeneralization

To keep the complexity manageable, and to simplify the software, it becomes necessary to reduce code duplication as much as possible across domains. This raises the question of when it is appropriate to encapsulate a new algorithm vs. refactor it to leverage a common infrastructure. 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 on 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 necessary to make the code repository complementary rather than duplicative.

Identifying and refactoring common software elements across domains reduces unnecessary code duplication. Proper class design, however, should avoid overgeneralizations, which leads to large abstractions that are hard to maintain. As capabilities and modules grow, one would refactor classes and split modules into smaller ones to maintain a manageable level of complexity.

Example: The Matrix class

To keep data structures maintainable, the Matrix class in *CLARAty* includes only basic operations such as matrix data management (inherited from N2D_Array), addition, subtraction, multiplication, and scalar operations. It does not include functions such as Lower-Upper decomposition, inverse, pseudo-inverse, Cholesky factorization, and singular value decomposition. These are implemented as separate matrix operations. We separate the *matrix* module that defines the structure and basic operations from the *matrix_op* module that includes these complex and mathematically intensive operations. This keeps the *matrix* module and the Matrix class light-weight and simple and allows an Image class to derive from the Matrix class, thus leveraging its basic matrix operations. So, if someone needs to use the Matrix data structure, they do not have to always incorporate the software that includes these mathematically intensive operations.

3.2 Algorithm Integration

Challenge: Different Programming Paradigms

There are two different programming paradigms that are used to develop intelligent robotic software: declarative programming and procedural programming. Declarative programming has dominated software developed by the artificial intelligence community while procedural programming has dominated software developed by the robotics community.

Declarative Programming	**Procedural Programming**
`Rover.navigate_from_to(Loc1, Loc2)`	**`If`** `near(Loc1,Loc2) AND`
`Preconditions:` `near(Loc1,Loc2)`	`rover.has_power(Loc1,Loc2) AND`
`rover.has_power(Loc1,Loc2)`	`rover.has_time(Loc1,Loc2) AND`
`rover.has_time(Loc1,Loc2)`	**`Then:`** `rover.navigate_from_to(Loc1,Loc2)`
`Effects:` `rover.is_at(Loc2)`	

Fig. 2. Declarative vs. procedural programming

Solution: Separate declarative and procedural programming into overlapping layers

These two programming paradigms are quite different for building robotic intelligence. In declarative programming, a programmer explicitly describes the activities, models, and constraints but does not provide any program logic (sequences, conditionals, and loops) that describes the order of execution. The program logic is automatically generated and updated by a search-engine that examines all constraints and maintains a plan to order activities without violating these constraints.

Conversely, procedural programming readily provides the program logic that contains the order of execution using activity sequencing, conditionals and loops. The execution flow is only altered through conditionals, exception, and dynamic binding. While declarative programming has infinite flexibility in ordering activities compared to procedural programming, it requires computational resources to generate the program logic and explicit constraints on all activities. In procedural programming, specifying the sequence of activities implies the order constraints. Fig. 2 shows a simple example using the two programming paradigms.

Because of their fundamental differences in formulating program logic, we recommend to separate and layer the two programming paradigms. The declarative programming portion of the software is often referred to as the decision layer while the procedural programming portion is referred to as the functional layer. Where the two layers meet continues to be an active area of research. An architecture where the two layers overlap provides the developers the flexibility to specify where they draw the line between the two layers. Because declarative programming is ideal for situations where the order of activity is less constrained (i.e. many activities can occur concurrently), it tends to dominate higher levels of the application software. On the other hand, procedural programming dominates mid- and lower-level software controls, which have a more constrained sequence of operations.

Challenge: Loosely- versus Tightly-Coupled Integration Model

Selecting the appropriate model for integrating algorithms largely depends on the nature of the algorithms to be integrated. There are several models for integrating robotic algorithms. Some models promote a looser integration where algorithms are encapsulated and wrapped into a framework, while other models promote a tighter model where algorithms are refactored to share a common infrastructure.

Solution: Support different levels for algorithm integration

Using a tightly-coupled integration model, where data structures are consistent, is more efficient and scalable than using a loosely-coupled model. In a loosely-coupled model where algorithms are encapsulated, data structures have to be converted to the format used by the algorithms. This is particularly difficult when memory is managed differently between the framework and the algorithms. Encapsulating algorithms results in redundancy and inconsistency in data representations among components, which can be subject to misinterpretations. It also leads to larger code bases, which can be harder to debug. However, using this model, algorithms are easier to develop at remote institutions since the model imposes fewer constraints on developers who can integrate subsystems fairly quickly for proof-of-concept demonstrations. But the resultant system is often fragile, hard to maintain, and does not scale well as the system evolves. Integrating using a loosely-coupled model is appropriate when the interface is small and the information to be exchanged is limited. The more sophisticated algorithms would require a tighter integration model and a shared infrastructure.

Using a tightly-coupled integration model where algorithms are refactored or implemented against a framework leads to more efficient implementations that are internally consistent and easier to debug. However, they require a commitment to a given framework making the algorithms framework dependent. Therefore, such a framework has to be widely accepted and available, mature and stable enough for developers to adopt and use.

Example: Advanced navigation

Consider a navigation algorithm that uses three-dimensional information from its stereovision sensors to select paths that avoid obstacles. An implementation of such an algorithm for indoor navigation would only need kinematic information about the robot (robot dimensions and types of maneuvers the robot is capable of). A more sophisticated version of this algorithm for use in rough outdoor terrain would also need to incorporate real-time dynamic information from the system. So, as algorithms increase in sophistication, the information and its flow get more sophisticated requiring a richer interface.

Challenge: Redundant Data Structures

Because robotics brings together many domains, software packages are often developed for these domains independently. Bringing together domain packages results in a duplication of data structures.

Solution: Maximize reuse through a cross-domain perspective on data structure classes

Take a global perspective to share common data structures to reduce unnecessary code duplication and reduce overall complexity. Common structure can be refactored into base classes that are cross-domain. Robotics software covers different domains such as motion control, locomotion, manipulation, vision, estimation, planning, scheduling, resource management, and health monitoring. The locomotion, manipulation, vision and estimation domains require similar math and coordinate transformation infrastructure.

Data structures can be general-purpose or domain-specific. General-purpose structures are reusable beyond the scope of robotics applications. The Standard Template Library provides an example of general-purpose data structures. Domain specific data structures include math (matrices, vectors, points), rotation matrices, quaternions, transformations (homogeneous and quaternion transforms), point clouds, paths, fuzzy sets, and so on. Decomposing software into entities that share a common infrastructure enables a more integrated, efficient, and consistent data flow across the system.

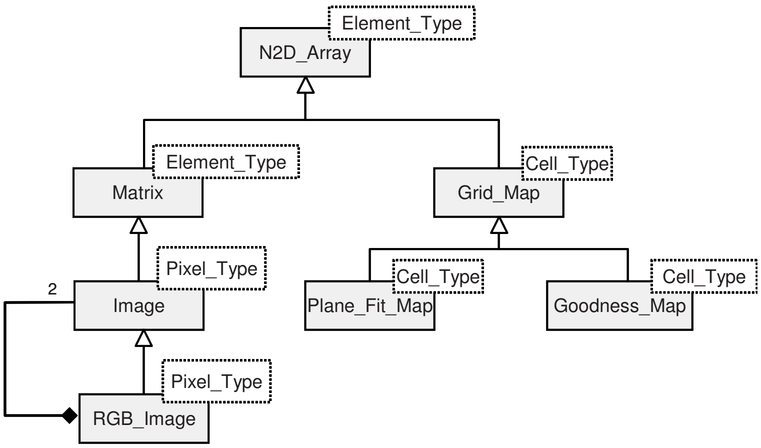

Fig. 3. Data structure abstractions

Example: The Array hierarchy

Fig. 3 shows the array hierarchy that is used in *CLARAty* . Several data structures use two-dimensional arrays. As a result, we define a base N2D_Array class to manage the contiguous storage of two-dimensional information. It provides functions to allocate, resize, retrieve rows and columns, access individual elements, manage sub-arrays, serialize and display the contents of the array. Array elements are not assumed to be numeric. A Matrix class extends the Array class to include numerical operations. Managing the storage of the Matrix elements is handled by the N2D_Array class. The Image class extends the Matrix class to include image specific functions such as pixel interpolation. Another set of classes that extend the Array class is the Grid and Plane_Fit_Map classes that are used for navigation. These classes are two-dimensional arrays whose elements are complex types: the Grid_Cell and the

Plane_Fit_Cell respectively. This example shows how the vision package, the navigation package, and the math package all share a common data type.

Challenge: Different Representations of Information

In robotics, there are multiple ways of representing the same information. Algorithms developed in isolation will most likely use different representations of information. 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.

Solution: Use generic programming with templates to support multiple representations efficiently

Because a framework integrates multiple algorithms from multiple sources, it needs to support different representations of information in its data structures. Data structures have to be efficient in dealing with multiple representations, so using polymorphism through inheritance to handle the different representation is neither efficient nor sufficiently flexible for mathematical types. Virtual inheritance adds the overhead of a pointer to the virtual table for every object and cannot support inline functions, which are critical for efficient operations. Instead, in these instances, use template binding to provide both the efficiency and flexibility. Façade classes [GHJV95] can then be used to provide a simple interface to the user to develop against. The principle here is to provide a wrapper façade class to simplify usage while keeping the underlying foundation flexible.

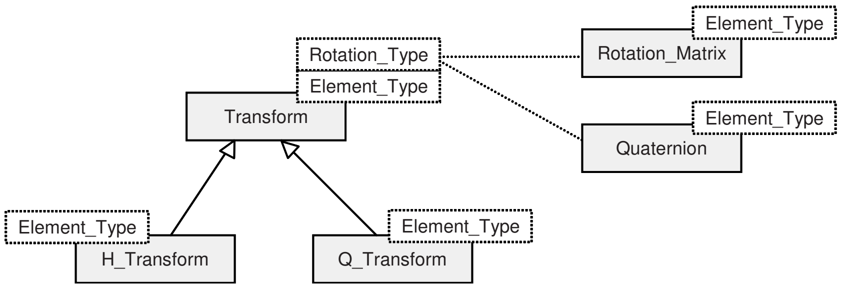

Fig. 4. Coordinate transformation classes

Example: Transforms

We can describe rigid body orientations using multiple mathematical representations such as: Euler angle, rotation matrices (*i.e.* direction cosine matrices), or quaternions. Euler angles are described by three floating-point numbers;

rotations by 3×3 matrices; and quaternions by a scalar and a 3×1 vector. 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.

Orientation is an integral part of a coordinate transform that consists of a translation and an orientation. Transforms can use any orientation representation. To represent that in software, we develop a transform as a template class of two types: rotation type and element type. The latter defines whether the elements use single- or double-precision floating-point numbers. Fig. 4 shows the relationship between the Transform class and the Rotation_Matrix and Quaternion classes.

A transform that uses a rotation matrix for its orientation is called a homogeneous transform. A transform that uses a quaternion is then called a quaternion transform. A homogeneous transform is mathematically represented as 4×4 matrix with the 3×3 rotation matrix and a 1×3 translation vector. Quaternion transforms do not have an equivalent mathematical representation. However, thanks to operator overloading, we can think of quaternion transforms as mathematical equivalents to homogeneous transforms. These specialized transform classes are derived from the template transform class by binding their rotation types to the corresponding orientation class: the quaternion transform binds to the Quaternion class and the homogeneous transform binds to the Rotation_Matrix class. This way both types of transforms share a flexible templatized core without exposing users to the complexities of the template implementation.

Challenge: Generalized Interfaces

Once the proper decomposition of modules and classes has been defined, the next challenge would be the design of the class interfaces. Defining proper interfaces is not a trivial task. Generic interfaces need to be stable and mature, but that can only be accomplished by exercising these interfaces across a number of heterogeneous robotic platforms over several years. It is the maturity of the class decomposition, inter-object communication and class interfaces that define the system's architecture.

Solution: Define comprehensive interfaces

Each class needs a complete set of interfaces for application developers to effectively use it. Minimal interfaces are often insufficient. Common base classes for certain types simplify the interface definition of many functions. Devices that represent physical components such as motors, sensors, instruments, and manipulators derive from a base Device class. This enables the connection of multiple instruments on a robotic arm where the end effector is of Device type.

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.

Example: Locomotor and manipulator interfaces

It is insufficient to provide only *move* functions for a wheeled locomotor class. Additional functions are necessary to control the overall speed and acceleration of the moves as well as stop the robot under normal and emergency conditions. We also need to have functions that access the locomotor's state (moving, stopped, or goal accomplished) and query for the actual speed and acceleration. All these functions are necessary to support the *move* capability. Further extensions may also include selecting the point on the robot's rigid body to control (as opposed to the default center of mass). They also include defining functions to move the rover along paths as opposed to goal-directed moves. Therefore, it is necessary to provide a complete set of interfaces to support a single functionality such as *move.*

One limitation of generic interfaces is the ambiguity they may yield. Consider the example of a four degree-of-freedom (DOF) arm where the three-dimensional position of the end effector can be specified but with only one degree of orientation. A generic manipulator interface would have to support the more general pose $(x, y, z, roll, pitch, yaw)$. If the generic interface is used to pass information to a limited degree-of-freedom arm, then a scheme has to be developed to handle the additional degrees of freedom. One may chose to report an error, ignore these extra terms or achieve the best that the limited DOF arm can do.

Challenge: Unstable Interfaces

Changes to the generic class interfaces reduce the stability of a framework and impact all applications that use the interface. Semantic changes can be checked at compile time, but behavioral changes are harder to manage.

Solution: Use complex data types to stabilize interfaces

While a complete stabilization of interfaces in a generic robotic framework may not be feasible, there is a strong need to minimize the impact of software changes on the generic interfaces. Using complex data types as opposed to primitive types significantly improves the stability of the interfaces. Complex data types hide the details of the implementation allowing the interface to be described using higher-level abstractions. The design of the data types, therefore, becomes of critical importance. The hierarchy of the data types determines what types can be used interchangeably.

There is no single data structure that dominates in a generic framework. While overgeneralizing interfaces to use a single common type provides the most flexibility, it defeats the purpose of a strongly-typed system. Strong type-checking is critical to ensure compatible interfaces and eliminate errors that result from the mapping of different primitive types. Without strong type-checking at the interfaces, changes to data structures cannot be properly managed.

Example: Camera interface

A Camera class has an interface to acquire an image. The interface can use the raw image data as follows:

```
camera.acquire(char* data, int nrows, int ncols)
```

or it can use an Image class to hide the details of the image implementation as follows:

```
camera.acquire(Image & image);
```

In the former case, adding a field to the image, such as image offset, impacts the acquire interface causing it to be changed to:

```
camera.acquire(char* data, int nrows, int ncols,
                           int srow,  int scol)
```

However, such a change does not impact the latter implementation because that change would be encapsulated in the Image class and will not be visible in the camera acquire interface.

3.3 Architectural Mismatches

Unless developed against a framework, robotic software components are likely to have architectural mismatches with the frameworks into which they will be eventually integrated. 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, we have an architectural mismatch. Similar situations occur when an algorithm requires high bandwidth information that may not be available on some platforms.

Another 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 optimize local performance, this often comes at the expense of global optimality.

Challenge: Mixing Units

Errors that result from mixing units can lead to catastrophic failures. When integrating heterogeneous algorithms from multiple institutions, it is important to pay careful attention to the inter-mixing of units.

Solution: Use a consistent representation of units

One possible solution is to develop algorithms and models that strictly use the *Systeme Internationale* (SI) units. This will certainly reduce the overall complexity, which is quite important in robotics. Allowing for a mix of units would otherwise require the use of tools and techniques to ensure proper unit conversion.

Several packages have been developed that use template-based classes to do unit checking at compile time. However, the use of unit conversion tools has some limitations and may give a false sense of assurance. This is particularly evident when dealing with larger data structures.

Another consideration for units is to keep the internal representation of units consistent, while supporting mixed units in the input and output files. This localizes the portions of the code that have to deal with mixed units to those handling file I/O operations, which frees up the rest of the software from having to deal with mixed units.

Example: The Image class

Consider a range image. It is desirable to have a single unit attached to an entire image as opposed to units for individual pixels, which would double the size of an image. However, when doing pixel-based operations such as projecting a pixel to its three-dimensional world location, that pixel needs to ensure that it retrieves the proper units from the image prior to any geometric computation. The complexity of adding compile-time unit checking and the risk that will result from the misuse should be weighed against the benefits of unit checking. But whether one chooses to adopt such a framework or use a units standard, that decision has to be made explicit.

Challenge: Representing Uncertainty

One of the primary challenges robots face is dealing with the uncertainty of the environment they operate in. Representation and reasoning about uncertainty is of primary importance. However, because different algorithms may use different representations of uncertainty, interpreting uncertainty becomes quite complex.

Another challenge relates to the assumptions and approaches that algorithms use. An algorithm for an industrial robotic arm, where precise motion and machined fixtures are expected, may handle uncertainty differently from that of an arm mounted on a rover operating in an unknown environment.

Solution: Use templates to represent uncertainty

To handle estimates and their uncertainties properly, use special template classes where the template arguments represent both value and its uncertainty as: `Estimate<Type,Uncertainty_Type>`. While this provides a data structure to capture the data representation, it does not address how this data is interpreted.

Functions that reason about system uncertainty, which are primarily used in estimation, need to use these template constructs to pass information around. Even though the majority of robotic applications use a Gaussian-based distribution, there are other distributions that algorithms use.

Example: Pose uncertainty

In its simplest form, a single variable such as a rover's heading can be represented by an estimate with a mean and a variance. This assumes a probability density function with a Gaussian distribution. Both the heading and the variance can be represented as single floating-point numbers. A rover's (x, y) position with cross correlation between its x and y values will have a 2×2 covariance matrix to represent the cross-correlation terms of x and y. So, an `Estimate<double,double>` is used for a single-valued estimate, and an `Estimate<Vector,Matrix>` is used for a vector and its covariance.

Challenge: Different Time Representations

Algorithm implementations may have different representations of time. They may also make different assumptions about information flow, which have to be reconciled with the framework.

Solution: Use abstract clocks and timers

To manage time, interface the software to clock abstractions as opposed to interfacing directly to the real-time clock. While it may not be quite obvious why managing time is critical for robotic applications, this becomes quite evident when robotic control software is interoperated between real and simulated platforms. On real platforms, it is natural to tie the system clock to the real-time clock. However, when the same software is run against a simulator, a number of options become available. Robotic control software has now the option to run faster or slower than real time. It can also control the stepping of a simulator's clock, enabling the use of otherwise computationally prohibitive control algorithms.

Timers attach to clocks and provide the ability to set, reset and advance time. Tying timers to Clock abstractions keeps time management consistent within a robotic system. Timers are used for measuring time intervals for trajectory generators and planners.

In robotic control applications, barring some exceptions[1], time can be largely used in a relative rather than an absolute sense. A robot needs to know the duration of its functions and activities. Absolute time comes into play when planning a day's worth of activities.

Representing time-values also require some attention. To ensure precise time representation, people use integers to represent absolute time. A single 64-bit integer is insufficient for absolute time measurements of microsecond precision over decades. Therefore two such integers are necessary. However, mathematical operations will require conversion of such integral values to floating-point numbers. Since the majority of time in robot control software is relative, a single integer or a double precision floating-point representation may be sufficient.

To avoid inconsistencies in time representation and to avoid improper mixing of real-time and simulated time, a framework needs to develop a consistent set of these time-based abstractions.

Challenge: Managing Periodic Activities

A single runtime model is often insufficient to handle all aspects of a robotics system. Robotic systems require both synchronous and asynchronous execution of different activities. Some activities have to run periodically at different frequencies. Activities need to be synchronized with other activities. Managing periodic activities can be challenging especially as its execution depends on the underlying operating system.

Solution: Use Periodic_Thread and Periodic_Object abstractions

Using Periodic_Thread and Periodic_Object abstractions provides a consistent way of handling periodic execution across a system. The Periodic_Thread (task) uses the aforementioned time-based abstractions in its implementation. It is responsible for creating a thread and scheduling the periodic execution of registered activities. It also checks the duration of each activity in order not to exceed the assigned interval. The Periodic_Thread class can only register active objects. Active objects inherit from the Periodic_Object base class, which defines a single pure virtual execute method. Active objects then implement this execute method.

The periodic thread uses the time-based abstractions and the thread abstraction to be independent from the underlying operating system. However, if the operating system is a soft real-time, then the periodic thread will only achieve the desired frequency on average. Recording execution statistics about the duration of the activities and the number of skipped activities provide necessary inputs on the performance of the system.

[1] Sun sensors are used to determine a rover's absolute heading using the sensed location of the sun in the sky together with ephemeris data and knowledge of the absolute time of the day.

A robotic system typically has a number of concurrent periodic threads. These include periodic updates to motor controllers, trajectory generators, closed-loop locomotor controllers, and pose estimators. These periodic threads may run at different frequencies depending on the application and hardware configuration.

Challenge: Runtime Models Are System Dependent

Runtime models define the processes and threads that are concurrently running in the system. Because of the variability in hardware architectures and because functionality can be migrated to embedded processors, the runtime models for software change with each platform. Furthermore, both the content and pathways of the information flow change with various device and system configurations, as well as with different application programs.

Solution 1: Separate generic from specific runtime models

It is necessary to separate the generic runtime models from the specific ones. Specific runtime models are system dependent. Therefore, it is necessary to encapsulate these models. The generic and specific runtime models should complement one another to maintain the same abstract behavior.

Example: Motor control

Some systems use motion control chips to servo the motor and generate local trajectories while others do so using software (Fig. 8). The behavior of the Motor class after attaching to either adaptation should remain the same. However, the runtime model for each will be different. In the first case, where the servo loop and trajectory generation are done in hardware, the motor adaptation communicates with the controller chip to send trajectory parameters, set control parameters, and retrieve motor information. In the second case where the motor control is done in software, an additional thread is necessary to periodically compute the trajectory set-points and servo control outputs. The control rate is set by the user.

While these two systems have different threading models to match their control architectures, their abstract motion control behavior should remain the same. The implementation details would be hidden from a person trying to develop a manipulator or a locomotor.

Some runtime models, however, can be generic. Consider the motor state machine shown in Fig. 5. This state machine is generic for all controlled motors. It is a hierarchical state machine with parallel states for the motor. The state diagram shows that the motor can either be in a MOVING or NOT MOVING state and at the same time be in a SERVOING or NOT SERVOING state. For instance, the motor can be in the NOT SERVOING state and the MOVING state at the same time if the motor moves as a result of external forces such as gravity pulling on a robot placed on a slope. From any

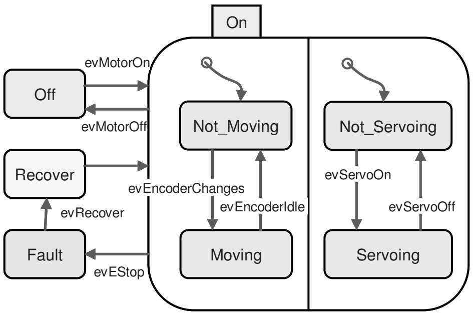

Fig. 5. Motor state machine

of these states, the motor can go into a fault state, which is followed by a transition to the recovering state before resuming normal operations. Such an implementation resides in the generic Motor class and describes the operation modes of the controlled motors. Adaptation of the generic Motor class can extend the state machine to include the specialized modes for the particular hardware component.

Solution 2: Encapsulate runtime models

It is not uncommon to off-load the processing of a computationally intensive algorithm to a separate processor. So, what does this mean for the software system? Since the component interactions are through class interfaces and since the run-time models are encapsulated, we can relatively easily off-load the implementation without impacting the rest of the software.

Example: Remote terrain analyzer

Fig. 6 shows the navigator class, which is one of the robot's behaviors. This class aggregates a locomotor and an action selector. The action selector can either use a grid-based representation of the world or a vector-based one. The grid-based representation that uses a terrain evaluator is computationally intensive because it operates on large data sets. As a result, the grid-base selector is a good candidate to off-load to a separate processor. The framework then implements a specialized grid-based selector that is a proxy to the real implementation. This proxy class is responsible for transferring the terrain and mechanism information to a remote grid-based selector, which in turn computes the data and returns the information to the proxy. The proxy and remote objects can use any communication protocol to exchange the data. However, the implementation is encapsulated such that the grid-based selector and the rest of the navigator classes do not need to know where the processing is taking place. Migrating functionality from one processor to another will be seamless. This is true with the exception of the initialization step that will require a definition of where computation occurs.

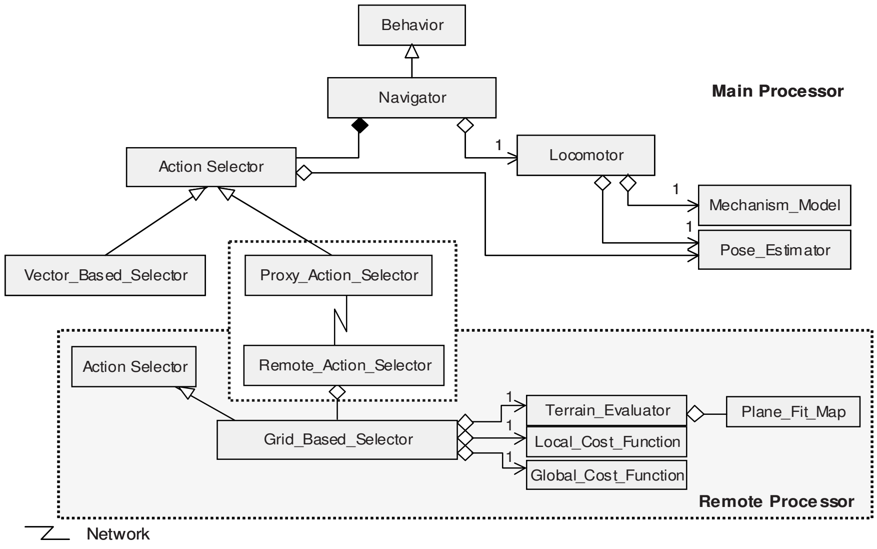

Fig. 6. Navigation abstractions

Challenge: Synchronous and Asynchronous Models

The primary challenge that arises between centralized and distributed systems is in the nature of information flow. In a centralized system, information flow is easily synchronized, while in a distributed peer-to-peer network, information flow and processing is asynchronous. A distributed system in a master/slave configuration is closer to a synchronous system.

Solution: Enable forward processing with constructs to synchronize activities

To ensure a responsive system, device objects must be thread safe and functions must support non-blocking modes. Move functions that do not block enable resumption of processing (*i.e.* forward processing) during a motion. However, it is necessary to provide control over the forward processing. Therefore, we need a function that can block on conditionals such as the percent completion of a move.

Example: Motor thread model

Fig. 7 shows two possible runtime models for the Motor class. The function `wait_until_done(%)` is used to control when the rest of the sequential processing can continue. In the first scenario, the motor is asked to change its position immediately, then wait until the motion is 45% complete before continuing to process that thread. At that instance, the thread blocks until that condition is met and then resumes execution. The motor continues to move while the motor class reads the current position. In the second scenario, there

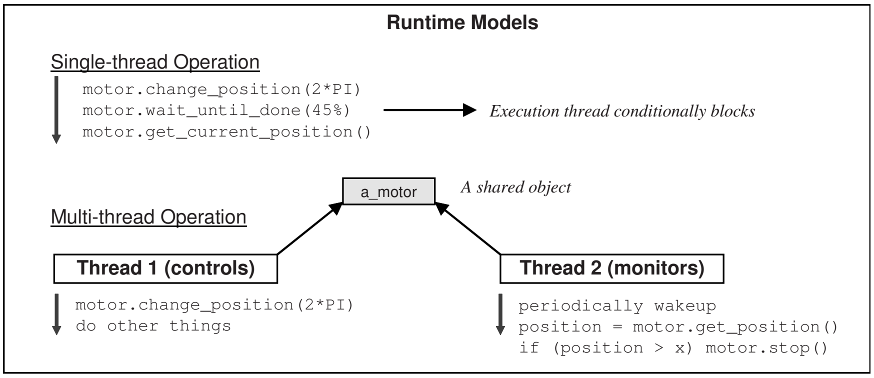
Runtime Models
Single-thread Operation
motor.change_position(2*PI)
motor.wait_until_done(45%)
motor.get_current_position()
Execution thread conditionally blocks
a_motor
A shared object
Multi-thread Operation
Thread 1 (controls)
motor.change_position(2*PI)
do other things
Thread 2 (monitors)
periodically wakeup
position = motor.get_position()
if (position > x) motor.stop()

Fig. 7. Motor run-time models

are two threads that communicate with a single instance of the class. The first thread issues a change in position and continues processing other functions. The second thread, which runs in parallel, queries the same Motor instance for the current position. The Motor class supports this parallel interaction and manages the communication link that ties it to the physical hardware.

Challenge: Shared Resources

A robotic system has a number of shared resources that need to be managed. Different resources may need to be managed differently depending on the urgency of the request for these resources. Shared resources in a robotic system include all sensors, actuators, hardware buses, digital and analog I/O, power, memory, and computational time. Multiple applications (clients) require concurrent access to shared resources for controlling various aspects of a robot.

Solution: (1) Protect data integrity using guards; (2) use reservation tokens to manage devices; and (3) manage activities with a global planner

Concurrent access to data requires the proper use of guards to protect data integrity. One can use several mechanisms for multi-threaded inter-object communication. Critical sections are easy to implement but can only be used when the protected operations are very limited in scope and are time bound. Critical sections disable system interrupts and can impact the responsiveness of the operating system. For small data structures, message queues are used. They require a copy of the data, but for a single consumer, such an overhead is acceptable for relatively small data structures. However, this can be quite costly for larger data structures such as images. In such instances, private member data are read and written using proper thread-safe read and write guards.

The above mechanisms can be used to manage resources at small time-scales that require fast context switching. However, using a resource for larger time-scales requires additional flexibility for managing these resources. For large time-scales, one can use reservation tokens where only one client can control a particular device. Only the client that has the token can command that device. However, multiple clients can query the device for information. Overriding reservation tokens is also necessary for handling emergency conditions.

Example: Shared camera

Consider a rover with two stereo camera pairs: one mounted on the front of the rover body while the second is mounted on an articulated pan/tilt mast head (see Rocky 8 in Fig. 1). The rover can acquire images and track targets from the articulated mast head independent of what the rover navigator does. These two applications use different cameras and mechanisms (mobility vs. mast pan/tilt). However, both algorithms share a FireWire bus and a motion control bus. Managing these shared resources efficiently provides a system where the two applications can operate in parallel.

Additionally, algorithms that operate in high-speed robotic system running concurrent activities must address computational latency to ensure correct behavior. A high-speed mobile robot running visual pose estimator in continuous mode needs to handle the latency between the time the data is acquired and the time the estimate is computed. In such situations, an additional step is necessary to compensate for the changes in rover state to produce the most accurate information about the robot's pose.

3.4 Software Efficiency

Challenge: Software Efficiency

Developing efficient algorithms using a generalized framework can be quite challenging. No one wants to trade performance for generality. Generality and flexibility may seem at odds with performance and memory efficiency. Despite the continued increase in available computational power and communication bandwidth, it is necessary to keep the performance of the generic software as close as possible to a custom solution.

Solution: (1) Use common data structures; (2) use templates for math; and (3) use inline functions

An application framework must pay particular attention to avoiding unnecessary copying of data when exchanging information among modules. This is particularly important when using component/connector style interfaces. The framework can also avoid accidental complexities that arise from isolated developments, which require transforming data from one form to another. Many techniques such as common data structures, the use of templates for low-level classes, and the use of inline functions can provide abstractions without the run-time overhead.

3.5 Multiple Operating Systems and Tools

Challenge: Hard versus Soft Real-Time

Many challenges stem from the differences in the runtime architecture of processes and threads that are used in current operating systems. Furthermore, some operating systems provide hard real-time scheduling guarantees while others provide soft real-time performance.

Solution: Use standard tools for operating system independence

To make the robotic software portable, it is necessary to build the software for different hardware architectures, operating systems, and compilers. This process also improves software reliability by eliminating nuances specific to an architecture, an operating system or a compiler. Third party packages such as POSIX and ACE (Adaptive Communication Environment) develop standards that cope with differences in operating systems. Proper handling of little and big-endianness, and the use of ANSI standards for compiler support ensure robust and portable software. However, the cost of developing and maintaining portable software has to be taken seriously into account. Another important aspect of this is the ability to transition from hard real-time operating system such as VxWorks, RTAI Linux, or Integrity, to soft real-time systems such as Linux, Solaris, Mac OS X, or Windows.

4 Hardware Variability

4.1 Hardware Architectures

Challenge: Different Hardware Architectures

While at first, it may seem that adapting abstractions directly to hardware components is all that is necessary to interoperate across robotic platforms, this assumes that the hardware on the various platforms is similar in architecture. This is rarely the case. Most robotic systems have different hardware architectures. On one end of the spectrum, there are robots that use a central processor to generate the motor control laws, motor trajectories, and run the application software. Such systems have their analog and digital signals mapped to memory registers on the central processor, which makes the development of software relatively easy (see Fig. 8). They are very flexible for changing the control laws and coordinating motors, but they lack hardware modularity and can be hard to extend and repair. On the other end of the

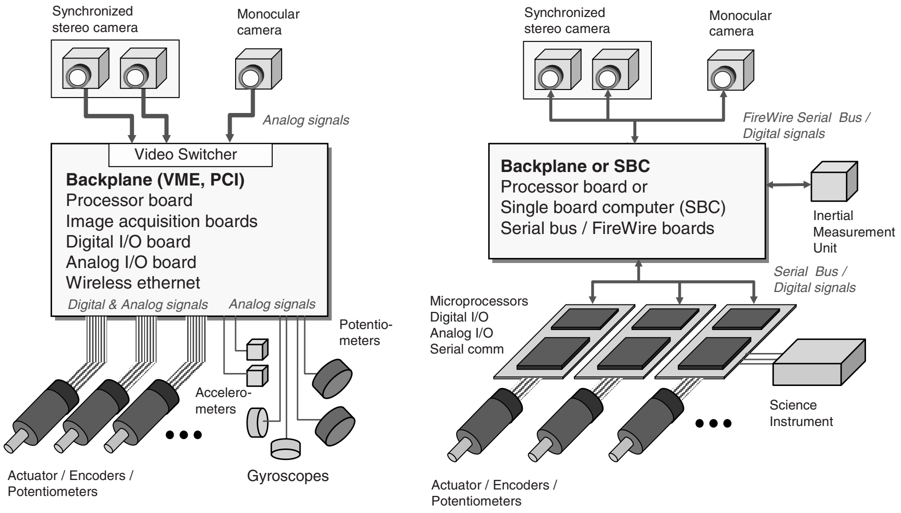

Fig. 8. Centralized (left) and distributed (right) hardware architectures

spectrum, there are systems that migrate much of their control to firmware in distributed nodes in order to improve modularity and reduce the load and real-time requirements on the central processor. They communicate with each other via formatted data streams sent over serial buses. Other systems fall somewhere within this spectrum.

There are similar differences in image acquisition and inertial sensing hardware. Some systems use high-quality analog cameras with centralized image acquisition boards (frame grabbers) while other systems use digital cameras connected through a serial bus (FireWire or USB). Inertial sensors can either be analog sensors integrated through a central processor or an integrated unit that communicates to a processor through a serial interface. A general framework must be sufficiently flexible to handle these variations in the hardware control architectures, which has significant impact on information flow and synchronization.

Solution: Use hierarchical multi-level abstractions

Develop multi-level abstractions to allow adaptations to interface to hardware at different levels. Different robots have different levels of hardware sophistication where some local intelligence may be embedded in distributed microcontrollers. Others may have inaccessible embedded processors limiting access to only higher-level interfaces for controlling the robot. Simulations also need similar multi-level access since each simulation may only support certain fidelity levels. Some may have full dynamics and device simulation capabilities while others may only simulate kinematics and approximate device models. In some cases, it may also be desirable to lower the fidelity level to speed up the simulation. For example, in a rover simulation, one may interface at the

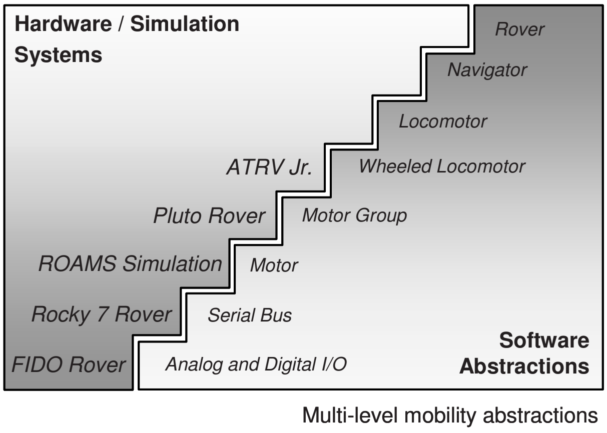

Fig. 9. Multi-level software access

locomotion level to simulate rover kinematic motions bypassing the need to simulate wheel dynamics.

Examples: (1) Multi-level access to a locomotor; and (2) migrating functionality to hardware

Fig. 9 shows a locomotion example that provides access to hardware and simulation at various levels of abstraction. At the lowest levels, the control software interfaces to hardware or simulation using basic analog and digital I/O signals. A higher level would be to interface through a serial bus where information is exchanged through formatted data packets as opposed to toggling I/O signals. At an even higher level, the software interfaces to a motor adaptation that understands motor commands. Higher levels include an interface to a group of coordinated motors, a wheeled locomotor, or a general locomotor.

Such examples not only occur with custom-built robots, but also when interfacing to commercial-off-the-shelf robots. Consider a commercial robot that does not provide access to individual motors. The manufacturer may provide a software layer with an interface to only control the robot's motion. Thence, the concept of a motor is hidden in a layer that is inaccessible to the user. Such a system will have adaptations at the locomotor level providing only access to the locomotor interface as opposed to the motor or I/O level interfaces.

To illustrate the point of migrating functionality into hardware, consider a robot with the stereo camera pair. This robot can acquire images synchronously from these cameras. A stereo processor class takes these camera images as input and outputs a depth map. This algorithm is implemented on the main processor and uses the images acquired by the cameras to generate a third depth image. While the stereo processor component in this example is a purely software component that does not interact with hardware, another stereo processor component on a different system can achieve the same capability in hardware. In the latter case, the stereo processing capability is migrated to a microprocessor that directly interfaces to the cameras. The stereo processor software component, in this case, is merely an adaptation to a hardware device while in the former case the hardware adaptation is at the camera level as opposed to the stereo processor level.

Challenge: Hardware Devices with Multiple Functions

What often complicates matters is when a hardware component provides multiple orthogonal functionalities. This is often the case when a robot system uses distributed processors. A capable micro-controller as the one shown in Fig. 8(right) not only provides motion control but also provides general purpose analog and digital I/O. To complicate things further, some of these analog and digital channels can be connected to other devices and instruments in the system. What we end up with is a system where devices that are logically decoupled become physically coupled. In such cases, we have a shared resource between multiple devices but which exists as a result of a given hardware architecture. We have to handle the physical dependency in a transparent way to keep the logical operation of devices independent.

Solution: Separate the logical architecture from the physical architecture

Decouple the logical architecture of the system from the physical hardware architecture. The physical architecture describes how the hardware works, while the logical architecture defines what the generic abstractions expect to have. Instead of combining these two into a single hierarchy, we recommend to separate them. An adaptation of the logical hierarchy would then bridge logical hierarchy to the physical hierarchy making the mapping between the two unambiguous. This adaptation class often contains little code but clearly shows the logical to physical mappings of functions. Mixing the two can lead to a single adaptation that is hard to understand, difficult to maintain, and hard to use in a specialized application that does not need the generic interface. Developing a hierarchy that is purely hardware specific keeps these drivers independent of any logical hardware architecture. It also makes the hardware specific code easily testable and more portable. This pattern applies to motors, cameras, digital and analog I/O, instruments, sensors and other actuators.

Even though the hardware architecture imposes constraints that result from shared resources (e.g. sharing a bus or a processor), the software has to manage these resources such that the overall behavior results in an independent logical architecture. To clarify this point, consider an operating system that provides a multi-tasking environment. The operating system has a shared resource: the processor. However, it provides the logical functionality of parallel tasks even though at its core it manages a shared resource.

Constraints that result from shared resources are managed locally by specialized classes. This decouples higher-level software that controls logically independent devices from the physical classes. The coupling is handled by the specialized classes, which allocate and free up these resources.

The assumption in the above model is that these resources are needed for only short durations. There are situations, however, where the robotic hardware may have severe constraints and cannot provide a logically decoupled architecture. In such situations, the software can compensate to the best of its ability for the missing or limited hardware functionality. This is similar to graphics acceleration cards where software compensates for missing hardware acceleration functions. Even though the overall performance will degrade, the software continues to operate. A similar example occurs in robotic motion control. If a motor controller cannot generate a motor trajectory to satisfy the requirements of an application, instead of sending trajectory parameters to the controller, the application software sends lower-level motor set-points to generate the desired trajectory.

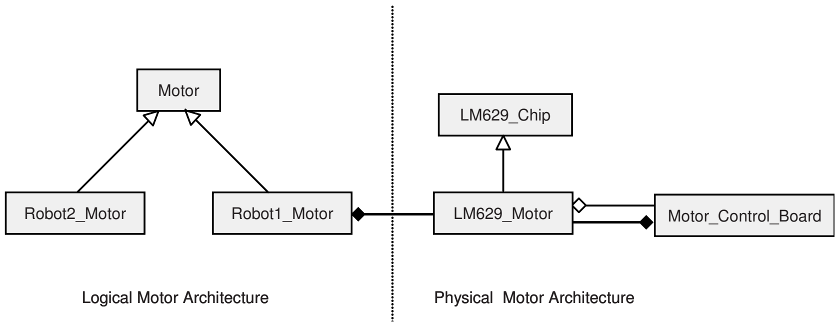

Fig. 10. Logical architecture vs. physical architecture

Example: Motor logical vs. physical architecture

Fig. 10 shows the two class hierarchies. On the left hand side is the logical hierarchy, which defines the generic motor functionality. While some functions will be pure virtual, others will provide a default implementation. On the right hand side is the physical architecture. In this example, the robot uses the LM629 motion control chip. The software driver for this chip can be made generic by keeping the functions that read from and write to the chip pure virtual. This enables the chip class to be specialized for different hardware boards. The LM629_Motor class that uses this chip in a given hardware board defines the communication interface. The Robot1_Motor class, which inherits from the Motor class, aggregates the LM629_Motor controller. The Robot1_Motor maps the physical LM629_Motor class to the generic Motor class. Because the LM629_Motor class does not depend on the Motor class, it can be tested independent of the generic hierarchy.

4.2 Hardware Components

Challenge: Class Design

Defining the proper classes, the interaction between these classes, and the organization of these classes into modules is difficult because different hardware components exhibit different behaviors.

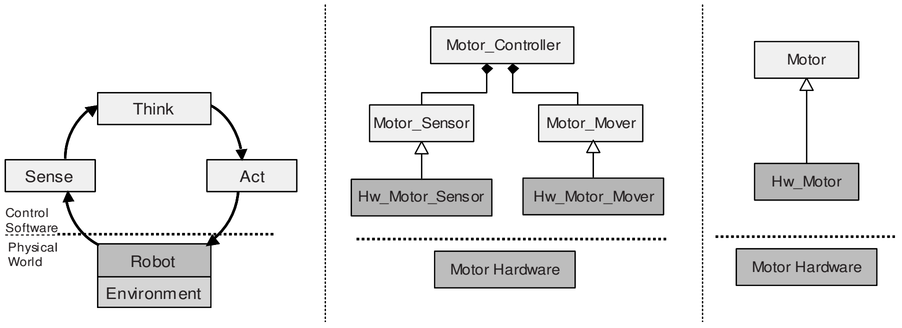

Fig. 11. Motor abstractions

Solution: Separate the specification of "what" to do from the "how" to do it

The classical approach in a robotic design is to separate the actuation from the sensing (see Fig. 11). Using this approach, a framework would have a Motor_Mover class and a Motor_Sensor class. The Motor_Mover would command the motor to move while the corresponding Motor_Sensor would read the encoder/potentiometer feedback and convert the motor shaft position. Another approach is to combine the two abstractions into a single Motor class that defines the capabilities and behavior of a controlled motor. These two approaches are fundamentally different. The first decomposition does not abstract a capability but exposes elements of the motor control. But the second abstracts the capability for controlling motors with a well-defined interface to enable the specialization of this capability. The Motor class provides only a partial implementation of its functionality and behavior. The remaining functionality is implemented inside specialized motor classes that adapt the Motor class to hardware components or to device drivers. Adaptations of motor cover different types of controllers including servo and stepper controllers, hardware and software controllers, and controllers with different sensory feedback such as optical encoders, magnetic encoders, or potentiometers. What is abstracted is not how the motor is doing the control, but rather what functionality a controlled motor provides.

Challenge: Generalize Low-Level Hardware Components

The main challenge in generalizing low-level hardware elements is maintaining the efficiency of a custom implementation.

Solution: Generalize hardware classes for greater flexibility

Even some hardware specific components can be properly generalized to provide the necessary flexibility to replace individual hardware components for low-level interoperability. One may consider digital and analog I/O as hardware specific components that cannot be generalized or, if generalized, become inefficient. Quite to the contrary, a large portion of digital and analog I/O control code can be generalized with proper abstractions, inline functions and templates to ensure a flexible and efficient implementation. Much of the digital I/O software involves bit manipulation and masking, which is independent of the hardware.

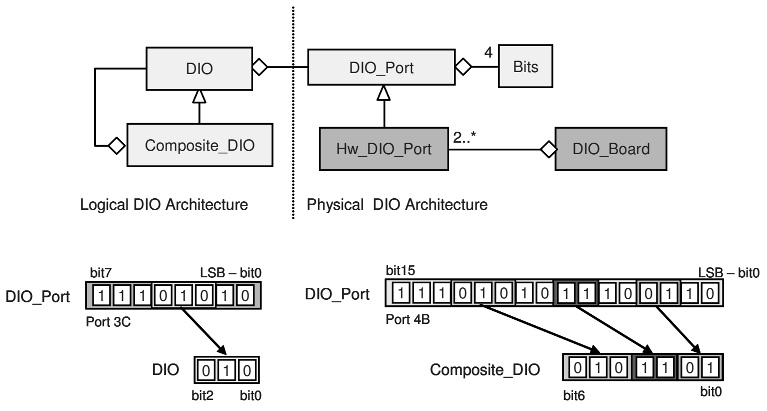

Fig. 12. Digital I/O abstractions

Example: Digital I/O class Hierarchy

Digital I/O provides another example where we separate the logical from the physical architecture. Generic digital I/O classes provide the logical mapping of I/O lines to the various instruments. The mapping of the I/O lines to the physical I/O ports is done through the hardware-based DIO_Port class. The DIO_Port class defines methods for input, output, configuration, and masking that are pure virtual. This class serves as a base class from which the hardware specialized ports are derived. Digital I/O boards often consist of a number of hardware ports. These ports contain a number of I/O lines that can be configured individually or as a group depending on the particular hardware board. These I/O lines can be configured as input, output, or tri-stated as both input and output. Fig. 12 shows the main DIO class, which aggregates a single DIO_Port. This pattern is known as a bridge pattern [GHJV95], which enables both the DIO class and the DIO_Port class to be specialized. This allows inheritance along two axes: the functional (logical) axis by extending the DIO class and the hardware axis by extending the DIO_Port class. Higher-level software now has the flexibility to control the digital I/O without being tied to any given hardware board or driver. With this design, we can replace any digital I/O board and re-map the I/O lines without changing to the application software.

The DIO class handles only contiguous bits. There are situations, however, where non-contiguous groups of bits need to be grouped and controlled synchronously. We handle these using the Composite_DIO class that both inherits from and uses the DIO class. The Composite_DIO class treats disparate DIO lines as a contiguous block of bits. It manages the splitting and grouping of bits to reflect the physical mapping of these I/O lines to hardware ports. Inheriting from DIO forces the Composite_DIO to be of the same DIO type making their objects interchangeable. Aggregating the DIO class enables multiple DIO objects in the Composite_IO. This pattern is known as a composite pattern [GHJV95].

Challenge: Class Flexibility and Extendibility

Because we are proposing to develop a generic framework, the classes and their interfaces need to be sufficiently flexible and rich in functionality to support the different use cases. This can lead to more complex class hierarchies that can be hard to extend and maintain.

Solution 1: Balance flexibility with maintainability

Despite the need for generality and flexibility, it is sometimes necessary to sacrifice flexibility for simplicity and improved maintainability. The challenge is in defining the proper level of flexibility to match the requirements without adding complexity.

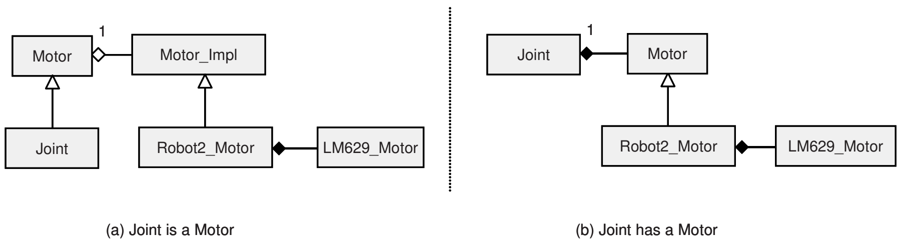

Fig. 13. Extending motor class functionality

Example: Joint specialization of Motor

Consider the motor example that we have presented earlier. The generic motor class gets specialized to a hardware adaptation. By specializing the motor to hardware, we can no longer extend its functionality. If we derive a joint from a motor, we then have to specialize the joint to the same hardware adaptation, thus duplicating the motor hardware adaptation. We can solve this problem using the bridge pattern [GHJV95]. Using this pattern, a new implementation class called Motor_Impl is created as shown in Fig. 13(a). The Motor class aggregates the Motor_Impl class. Now both Motor and Motor_Impl can be extended. A joint can now be derived from Motor to provide limit checking on the joint motions. This way a robot can use the unrestricted Motor objects for its drive wheels and the motion-constrained Joint objects for its steering.

While this seems like a reasonable solution to the problem, it has some drawbacks. First, all state information has to reside in the Motor_Impl class to be accessible to its adaptations. The Motor class would then become an abstract class. Second, every time we add a new function to the Motor_Impl class, we also have to add the same function to the Motor class. This structure can be hard to maintain if all devices in the system use this pattern. Third, this becomes particularly difficult when the Motor class derives from a generic Device class and the Motor_Impl class derives from the Device_Impl class.

In such cases, it may be easier to restrict device classes, such as motor, camera, and IMU from functional extensions. In this case, a Joint class has to aggregate a Motor object as opposed to inherit from it as shown in Fig. 13(b). What is lost here is that a Joint object is no longer of type Motor because it does not inherit from the Motor class. The Joint class would have to then redefine the functions that need to override the motor moves with ones that contain limit checking. The Joint class can also return a reference to the Motor to access to the rest of the functionality.

Solution 2: Group hardware devices into hardware maps

Because different deployments of generic robot software require different hardware maps, it is important to group the hardware components into a separate class that manages the system's configuration. An abstract factory class [GHJV95] can serve as a robot device map. A specialization of the device map creates the appropriate objects using the hardware specialized classes. For example, if the software is deployed on the physical hardware, then the motor, cameras, digital and analog I/O classes use their physical hardware components. If the same robot software is deployed in a simulation, then these devices would use their simulated counterparts.

Challenge: System State

System state is unique to each robot. In most complex systems, state is distributed throughout the system in its various microprocessors. The state information in these controllers can be retrieved, but there is often a limitation on the rate and the latency of this information. Some internal states might not be directly accessible.

Solution: Encapsulate states in the hierarchies

To support heterogeneous robotic platforms, state information should be handled in a hierarchical fashion. State would then be retrieved only through accessor functions as shown in Fig. 14. Similar to states, state machines should also be hidden in class abstractions. Breaking this encapsulation introduces system specific dependencies that reduce the interoperability of the software.

The states of a legged system differ from those of a wheeled system. However, the reason why software can be interoperated across heterogeneous systems is the presumption that there is a level of abstraction at which the generic algorithm can use abstract states to control the robot.

Example: Legged vs. wheeled locomotion

Consider two robots: a wheeled rover with a passive suspension and a legged robot (see Fig. 15). These robots can both be commanded to follow a path in rough terrain. However, the two robots will move differently along the path. The wheeled rover will conform to the terrain while the legged robot will articulate its legs to move its body along the path. While both robots follow the same path, they achieve that using different body motions. So at the generic locomotor level, both systems are commanded to follow a path even though they achieve that goal in different ways. The common states at the locomotor level are the robot's pose, the notion of a path, and the notion of

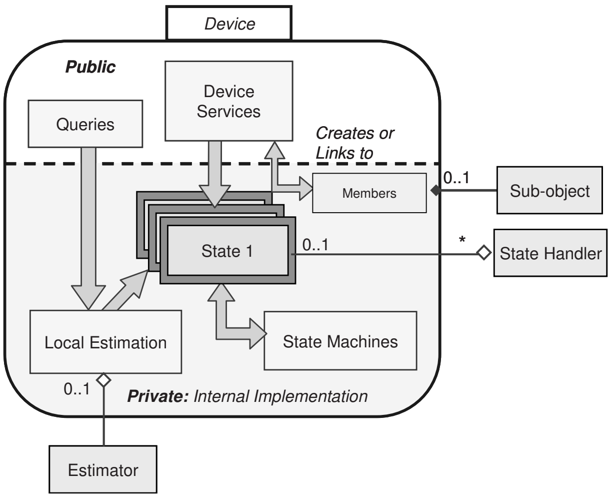

Fig. 14. Dealing with state and state machines

how far along the path a robot is. However, at the lower levels, the states differ. The wheeled locomotor keeps track of the state of each wheel (distance and steering angle) while the legged system keeps track of the joint angles for each leg. But the robot's pose, a higher-level state that is derived from these lower-level states, is common to both.

4.3 Sensor Configuration

Challenge: Different Sensor Configuration

The major challenge comes from differences in sensor configurations that are on similar physical robots. Similar robots may use different sensor configurations that produce similar information. However, different sensor configurations often have different physical constraints.

Solution: Use multi-level generic abstractions for the sensors in the system

By providing various levels of device abstractions, we can cope with the variability in the sensor configurations. This is particularly important as there are multiple implementations that may achieve the same result. Some may be available in hardware while others through software. This is best illustrated by the examples below.

Example: Stereo cameras vs. lidar; sun sensor vs. compass

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 robot. A stereo processing algorithm uses two images and their corresponding camera models to generate a three dimensional map. If what we are trying to do is to get a three dimensional map, then our algorithms should not depend directly on a stereo processor but rather on the three dimensional map or a point cloud source. A three-dimensional point cloud can also be generated from a lidar. So, by depending on the point cloud source as opposed to a stereo processor, our algorithms become generalized enabling them to work with a wider range of sensors. 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 example is with algorithms that use a compass. A more general algorithm would substitute its use of a compass class with the generic absolute heading sensor class. That way, a sun sensor, which also computes absolute heading, can be used interchangeably with a compass.

4.4 Different Mechanisms

Challenge: Different Capabilities

Different mechanisms exhibit different capabilities, which have major implications on how the robots move and act. Controlling and maneuvering wheeled robots is very different from controlling legged platforms. Even among wheeled robots, fully-steerable (omni-directional) rovers can move laterally (crab) while partially-steerable (car-like) robot can achieve the same result only via a parallel parking maneuver (Fig. 15(a) to (d)). Mobile robots with passive suspension conform to the terrain with no control over their tilt while those with active suspension have control over their tilt (Fig. 15(e) and (f)). Robotic manipulators have similar nuances. Joint configuration and degrees of freedom result in different constraints on the maneuverability of the end effector. Limited degree-of-freedom arms can only achieve certain poses, while redundant arms can achieve all poses with infinite possibilities.

Solution: Use multi-level model abstractions and separate models from control

Develop a generic model representation for mechanisms. Develop generalized kinematic and dynamic algorithms that use a generic model representation. Provide a means to override the generic capabilities with efficient specialized ones such as specialized forward and inverse kinematics.

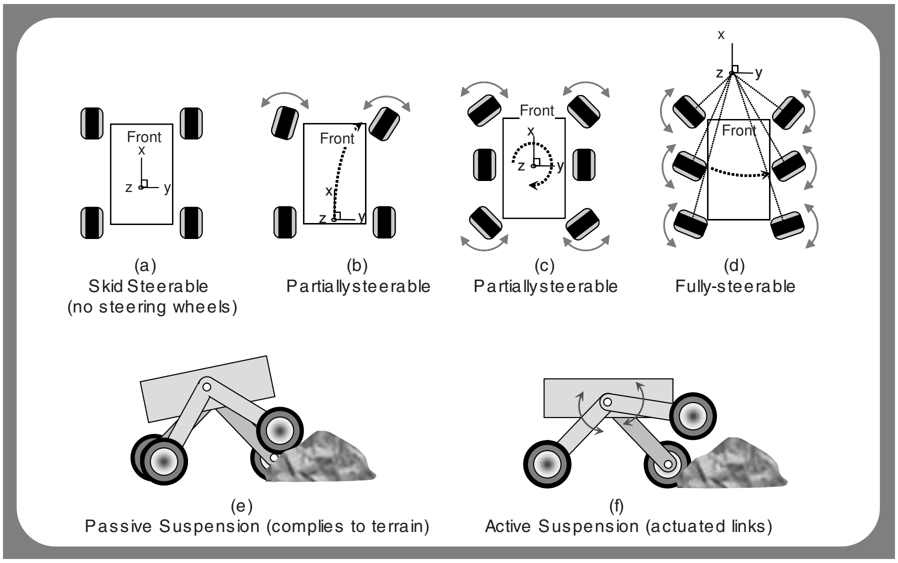

Fig. 15. Different mechanisms for wheeled robots

Separate mechanism models from controls to enable their use for resource and impact predictions. Embedding models into the same software structure as control makes their use outside that context very difficult.

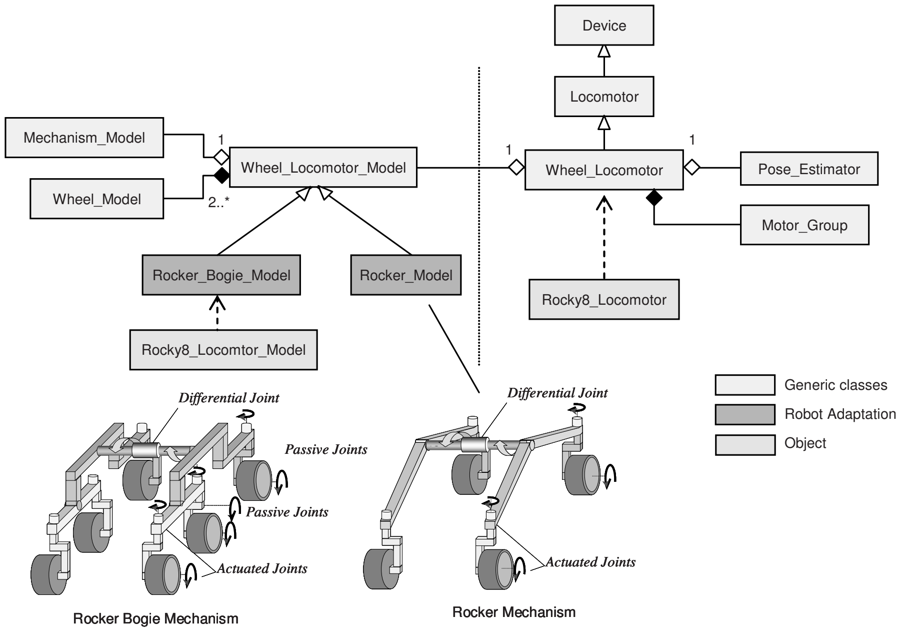

Fig. 16. Separating model from control

Example: Manipulator and locomotor kinematics

Fig. 16 shows the relationship between a generic Mechanism_Model and a Wheel_Locomotor_Model. The Wheel_Locomotor_Model aggregates a Mechanism_Model. The Wheel_Locomotor_Model provides generic capabilities for forward and inverse kinematics of all wheeled robots. It also includes wheel models classes. The Wheel_Locomtor_Model gets specialized to specific mechanisms such as a six-wheel rocker-bogie mechanism or a four-wheel rocker mechanism, which provide specialized kinematic solutions for their respective mechanisms. The Wheel_Locomotor control class aggregates the Wheel_Locomotor_Model class, which keeps the model hierarchy separate from the control hierarchy. Because of this separation, a navigator that requires information about the maneuverability of a vehicle now only relies on the Wheel_Locomotor_Model as opposed to the Wheel_Locomotor control class.

To illustrate the importance of separating models from control, consider a generic manipulator control class that does not separate the mechanism model from its control class. Rather, this class defines the forward and inverse kinematic interfaces in a specialized class that implements the closed-form inverse and forward kinematic equations. Using the forward and inverse kinematic algorithms now requires the instantiation of the arm with its motors and controllers even when only the kinematic portions of that class are desired for analyzing planned arm moves.

Challenge: Coordinate Transformations

Transformations cannot be defined in isolation and require a context that defines the relationships between coordinate frames. This is particularly challenging as some transformations go through articulated joints as shown in Fig. 17.

Solution: Unify mechanism model

Without a uniform representation of the robot mechanism, sharing mechanism information among sub-systems becomes difficult, inefficient and error-prone. A unified mechanism model is the backbone for managing coordinate frame transformations. It handles both fixed transformations and ones that go through articulated or passive joints. It also ensures the integrity of the mechanism information that is used by the multiple algorithms providing a consistent representation of kinematic, dynamic and geometric information.

A mechanism model will reduce code duplication when modeling robotic arms or mobility mechanisms. It also allows the development of generic algorithms for forward, inverse, and differential kinematics. In the absence of

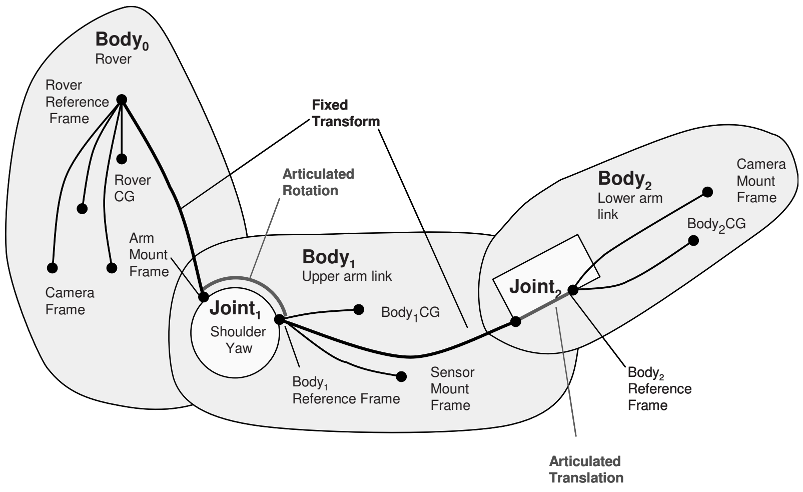

Fig. 17. Unified mechanism model

specialized versions, the generic algorithms provide out-of-the-box functionality. However, the architecture should support specific implementations to override generic algorithms whenever appropriate for optimal performance.

A mechanism can be represented using a tree topology in which an arbitrary number of rigid bodies are connected to one another via joints. The tree topology captures the geometric relationships between all elements in the mechanism such as sensors and bodies, and serves as a repository of mechanical model information. To support multiple clients the tree representation has to be stateless: position, velocity, and acceleration information relative to an inertial frame is not stored in the mechanism model. This enables various system states to be updated at different rates and enables the use of different parts of the tree at a time. It also allows algorithms to use the mechanism model tree to predict future states for any given input state. The trade that is made here is the cost of re-computing derived states vs. making copies of the mechanism model for each client application and keeping all their internal states up to date.

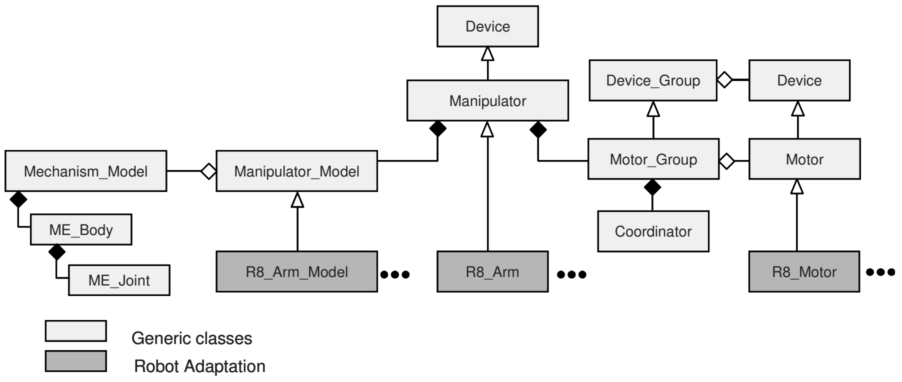

Fig. 18. Manipulator model and control classes

It is desirable for some applications to treat the mechanism as a whole body (for instance when dealing with rover-arm coordination algorithms). Some other applications may require the treatment of arms or legs as separate elements.

A mechanism model should support serial manipulators, closed-chains, wheeled mechanisms (Fig. 15), legged mechanisms, and composite mechanisms.

Example: The Manipulator classes

Fig. 18 shows the relationships between the bodies/joints and the generic mechanism model that are used by *CLARAty* . The manipulator model aggregates the manipulator portions of the complete mechanism model. A more detailed description of mechanism models can be found in [DCNKN06].

5 Conclusion

Many of these recommendations were adopted in the development of the *CLARAty* reusable framework that is used by NASA. 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.

Algorithms were developed at various institutions and have been integrated and tested on NASA developed robots. Capabilities that were integrated and demonstrated include motion control and coordination, wheeled and legged locomotion, stereo vision, visual tracking, visual odometry, science analysis, pose estimation, continuous trajectory following, path planning, autonomous navigation and obstacle avoidance, and general activity planning. We have also demonstrated autonomous end-to-end capabilities such as placing a rover-mounted instrument on a target selected from a 10 meter distance. 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 have been using *CLARAty* on half a dozen robotic platforms. Fig. 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. commercial 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 [Je04].

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 overgeneralizing 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 the least common denominator solution is often unacceptable. It is necessary to have flexible development environments, tools, and regression tests. Reusable software products and processes have to be well-documented. It would be highly desirable to standardize robotic hardware but that may not be feasible today.

There are many challenges in software engineering that any generic framework for robotics will have to address. No matter what approach is used in the design, issues related to the effectiveness of the framework can only be judged over time. The challenge is to find a delicate balance among flexibility, efficiency, scalability, maintainability, and extendibility.

Acknowledgement

The author would like to acknowledge the contributions of current and former members of the *CLARAty* team: Gene Chalfant, Caroline Chouinard, Dan Clouse, Antonio Diaz-Calderon, Tara Estlin, Daniel Gaines, Ron Garrett, John Guineau, Won Kim, Richard Madison, Michael McHenry, Michael Mossey, Darren Mutz, Hari Nayar, Richard Petras, Mihail Pivtoraiko, Gregg Rabideau, Babak Sapir, I-Hsiang Shu, and Marsette Vona from the Jet Propulsion Laboratory; Clayton Kunz, Lorenzo Fluckeiger, Susan Lee, Eric Park, Randy Sargent, Hans Utz, and Anne Wright from NASA Ames Research Center; Reid Simmons, David Apfelbaum, Kam Lasater, Nik Melchoir, Chris Urmson, and David Wettergreen from Carnegie Mellon; and Stergios Roumeliotis, Anastasios Mourikis, and Nikolas Trawny from the University of Minnesota. He would also like to acknowledge the contributions of former principal investigator Richard Volpe and former lead engineers Anne Wright and Max Bajracharya as well as the numerous contributors from universities and NASA centers who have provided algorithms into *CLARAty* .

The author would also like to thank NASA's Mars Technology Program and the Robotics management at NASA and the Jet Propulsion Laboratory for their vision and support: David Lavery, Samad Hayati, Paul Schenker, Richard Volpe, and Gabriel Udomkesmalee. The work described in this chapter was carried out at the Jet Propulsion Laboratory, California Institute of Technology, NASA Ames Research Center, Carnegie Mellon, and University of Minnesota under a contract to the National Aeronautics and Space Administration.

References

[ACF98] R. Alami, R. Chautila, S. Fleury, M. Ghallab, and F. Ingrand, *An architecture for autonomy*, The International Journal of Robotics Research **17** (1998), no. 4.

[Alb00] J. Albus, *4-d/rcs reference model architecture for unmmanned ground vehicles*, IEEE Internation Conference on Robotics and Automation (San Francisco), April 2000.

[AML87] J. Albus, H. McCain, and R. Lumia, *Nasa/nbs standard reference model for telerobot control system architecture (nasrem)*, NBS Technical Note 1235, National Bureau of Standards, Gaithersburg, Maryland, July 1987.

[cla06] *http://claraty.jpl.nasa.gov*, 2006.

[DCNKN06] A. Diaz-Calderon, I.A. Nesnas, W. Kim, and H. Nayar, *Towards a unified representation of mechanisms for robotic control software*, International Journal of Advanced Robotic Systems **3** (2006), no. 1, 61–66.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Villisides, *Design patterns: Elements of reusable object oriented software*, Addison-Wesley, 1995.

[Hat03] M. Hatig, *Robotic engineering task force*, IEEE Int'l Conference on Intelligent Robots and Systems (Las Vegas, Nevada), October 2003.

[HP86] V. Hayward and R. Paul, *Robot manipulator control under unix rccl: A robot control c library*, International Journal of Robotic Research, 1986.

[jau06] *http://www.jauswg.org*, 2006.

[Je04] A. Jain and et.al., *Recent developments in the roams planetary rover simulation environment*, IEEE Aerospace Conference (Big Sky, Montana), 2004.

[KT98] C. Kapoor and D. Tesar, *A reusable operational software architecture for advanced robotics*, CSIM-IFTiMM Symposium on Theory and Practice of Robots and Manipulators (Paris, France), 1998.

[NSG06] I.A. Nesnas, R. Simmons, D. Gaines, C. Kunz, A. Diaz-Calderon, T. Estlin, R. Madison, J. Guineau, M. McHenry, I. Shu, and D. Apfelbaum, *Claraty: Challenges and steps toward reusable robotic*, International Journal of Advanced Robotic Systems **3** (2006), no. 1, 23–30.

[NWB03] I. A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, and W. Kim, *Claraty: An architecture for reusable robotic software*, SPIE Aerosense Conference (Orlando, Florida), April 2003.

[PCSCW98] G. Pardo-Castellote, S. Schneider, V. Chen, and H. Wang, *Controlshell: A software architecture for complex electromechanical systems*, International Journal of Robotics Research **17** (1998), no. 4.

[SVK97] D. Stewart, R. Volpe, and P. Khosla, *Design of dynamically reconfigurable real-time software using port-based object*, IEEE Transactions on Software Engineering **23** (1997), no. 12.

[VG06] R.T. Vaughan and B.P. Gerkey, *Reusable Robot Software and the Player/Stage Project*. In D.Brugali (Ed.) Software Engineering for Experimental Robotics, STAR, Springer Verlag, 2006.