CLARAty:
Coupled Layer Architecture for Robotic Autonomy

Richard Volpe, Ph.D.
Issa A.D. Nesnas, Ph.D.
Tara Estlin, Ph.D.
Darren Mutz
Richard Petras
Hari Das, Ph.D.

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California 91109
Email: *firstname*.*lastname*@jpl.nasa.gov

December 2000

Abstract

This report presents an overview of a newly developed robotics architecture for improving the modularity of system software while more tightly coupling the interaction of autonomy and controls within the system. To accomplish this, we have modified the conventional three-level robotics architecture with separate and distinct functional, executive, and planning capabilities, into a new two-layer design. This design features a tight coupling of the planner and executive in one Decision Layer, which interacts with a separate Functional Layer at all levels of system granularity.

The Functional Layer is an interface to all system hardware and its capabilities. It has a number of specific characteristics. First and foremost it is an object-oriented hierarchy which captures granularity and abstraction of the system. Second, the state of all system parts is contained in the appropriate objects and obtained from them by query. This includes state variable values, object state machine status, resource usage, health monitoring, etc. Third, all objects have basic encoded functionality for themselves, accessible from within the Functional Layer, as well as the Decision Layer. Fourth, all objects may have local planners, such as those used for arm motion or driving. Fifth, objects have resource usage predictors, providing estimates of future resource usage to specified levels of fidelity, where high levels may require access to subordinate objects. Sixth, system simulation can be obtained at various levels of fidelity by substituting subordinate emulation objects for their hardware counterparts. Finally, objects contain test and debug interfaces and have external exercisers. The Functional Layer may be used directly for development, but is accessed only by the Decision Layer during autonomous operations.

The Decision Layer uses the capabilities of the Functional Layer to achieve objectives that it has recieved from operators. These objectives are used to build a Goal Net, which is a temporal constraint network at the planning level and a task tree at the executive level. Goals in turn are specified as constraints on state over time, consistent with the philosophy of the Mission Data System and ASPEN/CASPER planning systems. Goals essentially specify "what not to do". Tasks, on the other hand, are specified as explicitly parallel or sequential activities that are tightly linked. They are consistent with executive languages such as the TDL and are compiled in. Tasks essentially specify "what to do" within the limits of the goals. Tasks may be implemented by Commands, the termination fringes of a goal net where the Functional Layer is accessed. This lower border of the elaborated goal net, is called "The Line", and floats according to the current elaboration. When projected on the Functional Layer, it denotes the border below which the system is a black box whose behavior is well characterized. While this system is operating, the Decision Layer has access to the entire state of the system for planning purposes. State of the Functional Layer is obtained by query. State of the Decision Layer, which is essentially its plan, the active elaboration, and history of execution, is maintained by itself. Therefore, it may be saved or be reloaded, in whole or part.

During operation, the system first recieves and expands goals while obtaining resource predictions from the Functional Layer. Next is Scheduling, where rearrangement of activities is based on the resource constraints. With the time-line developed, execution proceeds. The Executive can also expand some activities into task trees or directly into commands, which access the Functional Layer. Execution of these activities, some of which may be conditional, brings state feedback and causes changes in resource usage values.

These changes are handled by Plan Repair, where iteratively repairs are made on the plan based on the new projections of resources.

In addition to a more detailed description of these concepts, we present examples within several contexts, and progress to date in implementing the initial version of this architecture. Initial efforts have utilized our research rover systems Rocky 7 and 8, and leveraged heavily on previous work in Planetary Dextrous Manipulators, ASPEN/CASPER, MDS, and Long Range Science Rover research projects at JPL, Caltech.

Contents

List of Figures

Part I

Overview

Chapter 1

Introduction to CLARAty

1.1 Background of this Effort

1.1.1 History Outside of JPL

The development of Robotics and Autonomy architecture is as old as the field itself. Therefore, it is not possible here to completely review the body of work upon which this effort builds. Instead, we will simply describe some of the more recent or dominant trends influencing the new architecture presented in this document.

Efforts in robotic architectures have largely arisen from a pragmatic need to structure the software development for ease of system building. As such, they have grown in scope and complexity as the corresponding systems have grown. Early efforts concentrated in detailed software packages [48], or general frameworks [6]. Only in the last decade, with the emergence of fast computers with real-time operating systems, have infrastructures been designed as open-architecture controllers of modern robot systems [76, 68, 19].

In parallel with robot control efforts, artificial intelligence systems for planning/scheduling and execution were developed which relied on underlying closed-architecture robot controllers [33, 70]. The tendency of these systems to be slow and computationally costly led to the emergence of a minimalist school of thought using Behavior Control [22]. But with faster control layers available, and a general desire to leverage planning functionality, newer systems implement a multi-tiered approach that includes planning, execution, and control in one modern software framework [5, 7].

While these end-to-end architectures have been prototyped, some problems have emerged. First, there is no generally accepted standard, preventing leverage of the entire community's effort. This problem has led to the second, which is that implemented systems have typically emerged as a patchwork of legacy and other code not designed to work together. Third, robotics implementations have been slow to leverage the larger industry standards for object-oriented software development, within the Unified Modeling Language (UML) framework. Therefore, we believe the time is ripe to revisit robotics and autonomy efforts with fresh effort aimed at addressing these shortcomings.

1.1.2 History Inside of JPL

The Jet Propulsion Laboratory, California Institute of Technology (JPL) has a long history in building remotely commanded and controlled spacecraft for planetary exploration. Most of this effort has concentrated on very simple and robust execution of linear sequences tediously created by ground controllers. Areas where expertise has concentrated on sophisticated on-board closed loop control have been largely outside of the traditional areas of robotics, falling instead in the realm of aerospace guidance and navigation. Further, the implementation of these solutions have been in hand tailored software solutions, optimized for specific spacecraft and limited CPU and memory. Only more recently have concepts from robotics and autonomy started to be used or considered for flight missions [61, 57].

Therefore, the history of robotic efforts at JPL has been primarily within the research program. The oldest of these efforts were in the areas of manipulator and teleoperation systems, and had limited software or software architecture components [14]. One of the first major software architecture efforts was within the Telerobotic Testbed, a large research effort for developing autonomous, multi-robot, satellite servicing [13]. While a very complex system conforming to the NASREM architecture [6], it relied on several subsystems using disparate software paradigms. Except through the diffuse efforts of the individual research participants and their subsequent assignments, little of this software structure survived the demise of the Testbed. Afterward, many smaller on-orbit manipulator research projects existed, each with their own software implementation: Remote Surface Inspection (C and VxWorks), Satellite Servicing (C and assembly), MOTES (Ada and VxWorks), etc. [80, 15, 11]. Each of these efforts provided parallel duplication of similar functionality with minimal code sharing due to architectural differences.

In parallel with these robot manipulation efforts were several mobile robot efforts, each developing software infrastructure in relative isolation. At about the same time as the Testbed, there was the development of a large Mars Rover platform name Robby, using C and VxWorks [83]. Research with Robby ended as there was a paradigm shift from large rovers with software for deliberative sensing and planning, to small rovers with reactive behaviors [42]. The fourth of these "*Rocky*" vehicles, programmed in C and Forth without an underlying operating system, sold the concept of placing the Sojourner rover on the Pathfinder Mission. However, Sojourner itself was programmed with software written from scratch, not inherited from its predecessors.

Only as Sojourner was being built did new rover research begin to address the problem of providing a software infrastructure with modularity, reconfigurability, and code re-use implicit in the design. To this end, a new rover, *Rocky 7*, was built, and its development team selected the ControlShell C++ software development environment hoping to set it as a new standard [68, 79]. But as subsequent research rover efforts were started, a new spectrum of control infrastructures re-emerged in rover tasks (e.g. ET FIDO, DARPA TMR Urbie, Nanorover, etc), similar to the situation seen in manipulation tasks half a decade before [67, 86, 77].

In the same time-frame as the construction of Sojourner and *Rocky 7* there was a large scale effort in Autonomy and Control for flight, but targeted for cruise and orbit, not surface operations. Under the aegis of the Deep Space One project and later renamed the Remote Agent Experiment (RAX) [61], this was a collaborative effort between JPL and NASA Ames Research Center (ARC). Emerging from it, was a determination at JPL to build a fundamentally new software architecture for all future missions, named the Mission Data System [27]. MDS is a state-based, object-oriented architecture that moves away from previous mission control concepts which are sequence-based. While it was originally targeted for orbital insertion and outer-planet missions, it is now addressing a Mars surface mission scheme for its first application.

Therefore, given the large efforts in software architecture development at JPL under the MDS flag, and given the history of divided efforts in the robotics research community, it is the objective of authors of this report to put forth a new framework for robot software at JPL and beyond. This report outlines the results for the first year, describing the broad design of the resultant CLARAty architecture, providing some initial implementation efforts, and outlining the directions for upcoming construction of end-to-end rover control software under this new framework.

1.2 The Challenge

Having briefly reviewed the history of robot control architectures, it is apparent that more work is required. In this section we will summarize the impediments to success that have existed in the past, outline the reasons for attempting to overcome them with a new architecture, and describe the constraints on the solution to be provided.

1.2.1 Impediments to Success

There are numerous impediments to the success of control frameworks for robotics systems. These may be categorized as follows:

Programmatic Vision: Implicit in the success of any research endeavor is the need to sustain the effort with funding, especially early in its development. Typically it has been difficult to maintain significant research funding for control architecture development. This is primarily because the end product is infrastructure, not a new robot system or algorithm. While this new infrastructure might enable better or faster system and algorithm development, such indirect results have been difficult to sell programmatically.

Not Invented Here (NIH): For the most part, autonomy and robotic systems are still in the domain of research products, and not commercial products. Therefore, it is typical for each research team to want to develop and grow its own products. This expresses their inventiveness, as well as giving their work a unique signature used in promotion of their results.

Fear of unknown products: Closely tied to NIH, is the fact that research products from outside of one's team have varying and unknown levels performance, quality, and support. Therefore, use of other's products might not only dilute one's research identity, but consume valuable effort while trying to adopt them.

Flexibility: Also, because of the research nature of robotics, there is still no absolute consensus on how to best solve the problems that exist, or even which are the most important problems to solve. Therefore, researchers often desire maximum flexibility from their hardware and software, to meet the specific needs of new projects. This desire for flexibility is often at odds with any software framework that is not specifically tailored to the task. Simply put, no one architecture can be optimal for all problems, and if one is too flexible it quickly loses any structure that gives it value.

Overhead: Often coupled to a desire for flexibility is a need to optimize performance. This comes in the form of computational overhead for the robot system, as well as system building overhead, encountered in the use of software development products which are unfamiliar or unwieldy.

Critical Mass: Even if a new software infrastructure is recognized as being valuable, that value might not be realized unless a large enough group of researchers chooses to standardize around it. Once such a group exists and provides critical mass, the standard enables much easier exchange of ideas and software, which in theory can "snowball". However, it is a difficult decision for any one research team to join a new standard until critical mass has been reached. This is because any external standard will require overhead, while the benefits may only come after critical mass is achieved.

Learning Curve: Human nature and conservative logistics of any research program provide a resistance to abandoning well known and understood methods for new ones that require an investment of time to learn. This is especially true when projects are on short development cycles, which has been more true in recent years.

Technical Vision: Because most researchers have had to develop infrastructure to build their systems, they have developed opinions about their preferred solutions. While many may be willing to abandon these solutions in favor of an external product, some will surely have a technical vision which is at odds with external products. Depending on the strengths of their convictions, these researchers may not join the larger community in the use of an external architecture standard. Independent of the implications for their own research, the loss of their participation is likely to be detrimental to the community.

1.2.2 Needs for a New Start

Given these impediments to the acceptance of a unifying architecture, one may wonder why there should be an impetus for its creation. The primary reason is that which drives the desire for robotics in the first place: elimination of the need for people to waste their time on lesser endeavors. There are three paths to this goal.

1. **Elimination of duplicative efforts which prevent attainment of critical mass:**

   Parallel Duplication: As previously discussed, there are often duplicative efforts within both robotic manipulation and mobility research. This diminishes the final products by wasting resources on solving the same problems, with different infrastructure, at the same time.

   Serial Duplication: It is also evident that as new research tasks start, they often wipe the slate clean to eliminate old system problems and lack of familiarity or trust with previous products. Typically, the only software with legacy is due solely to a single individual, not the local community. Obviously, without the ability to bridge to group ownership, transfer outside of institutions is even more restricted.

2. **Follow software community lead:**

   Open source movement: The value of shared software has been dramatically illustrated by Linux, GNU, and other share/free ware products. Typically this has existed within the desktop PC market, but there is no obvious reason why this model cannot be leveraged by software within the robotics community. As evidence of this fact, there has recently been an announcement for Intel sponsorship of an open source Computer Vision Library [49].

   Object-oriented design: Complementary to the open source movement, has been the growth of object-oriented design for PC software. In much of the commercial software industry it dominates. However, this paradigm is largely under-utilized in robotics, isolating the community. Further, it promises to better facilitate software sharing, discussed next.

3. **Leverage complimentary efforts:**

   Software sharing: To build critical mass amongst a world-wide but relatively small robotics community, it would be extremely beneficial to have an architecture framework that was widely accepted. Not only would this enable easier sharing of design concepts, but, more importantly, it would enable the direct transfer of software to all parties. Even sharing amongst the limited communities of JPL and NASA is currently arduous and therefore rare. A first step would be to eliminate these hurdles completely.

   Mission Data System and X2000: Recently, NASA has invested heavily in large scale efforts in spacecraft hardware (X2000) and software (MDS) which promise an infrastructure to be leveraged and expanded [84][27]. It is to the benefit of NASA robotics efforts to leverage these products where applicable. Since the spacecraft control problem is very similar to the general robotics problem, it is anticipated that there is much to be gained by this leveraging. Obviously other sources of relevant technology will exist outside of this limited set, and will also be incorporated when applicable.

1.2.3 Constraints on the Solution

Given these needs, there are several issues that will constrain the success of an architectural solution. First, there is a need for community acceptance. Without acceptance by the robotics and autonomy community, both from users and developers, there can not be a success. Full acceptance is probably not possible, or even desirable in a growing research area. However, as described previously, it is important to reach a level of critical mass, so that users and developers gain more than they lose from adherence to standards and participation in software exchange.

Second, it is vital to span the many divides within the necessary user and developer communities. These divides exist in many forms, between and within robotics and AI research areas. They can result from a desire to solve different types of robotics problems, all the way from parts assembly to humanoid interfaces. Or they can result from an emphasis on different phases of product life cycles, from basic research to fielded systems. Within and across institutions, the differences can be cultural as well, spanning departments from mechanical engineering to computer science, and organizations from academia to commercial companies.

Third, there is a desire to leverage existing software in research and NASA flight efforts. In particular, at JPL there has been a substantial effort in the new MDS, which is very similar to the architecture work described herein, but has been largely focused on the problems of zero-gravity spacecraft, not robots operating on planetary surfaces.

Finally, it is a requirement to leverage standard practices in industry. This is needed to avoid reinvention of the wheel, and enable NASA robotics efforts to adopt techniques and solutions commonly employed in commercial products, and within the global software community.

1.3 The CLARAty Architecture

In response to these needs and requirements we have developed the initial framework for a new Autonomous Robot software architecture. Due to its structure, it is call the Coupled Layer Architecture for Robotic Autonomy, or CLARAty. This section will review this new structure, and its evolutionary differences from its predecessors. It will introduce the two layers of the architecture and provide an overview of the interaction between them. Subsequent chapters will provide a much more detailed description of each layer.

1.3.1 Review of the Three-Level Architecture

Typical robot and autonomy architectures are comprised of three levels — Functional, Executive, and Planner as shown in Figure 1.1 [41, 71, 5].

The dimension along each level can be thought of as the breadth of the system in terms of hardware and capabilities. The dimension up from one layer to the next can be thought of as increasing intelligence, from reflexive, to procedural, to deliberative. However, the responsibilities and height of each level are not strictly defined, and it is more often than not the case that researchers in each domain expand the capabilities and dominance of the layer within which they are working. The result are systems where the Functional Layer is dominant [69, 81, 56], or the executive is dominant [71, 19] or the the planner is dominant [33, 30]. Further, there is still considerable research activity which blurs the line between Planner and Executive, and questions the hierarchical superiority of one over the other [52, 35]

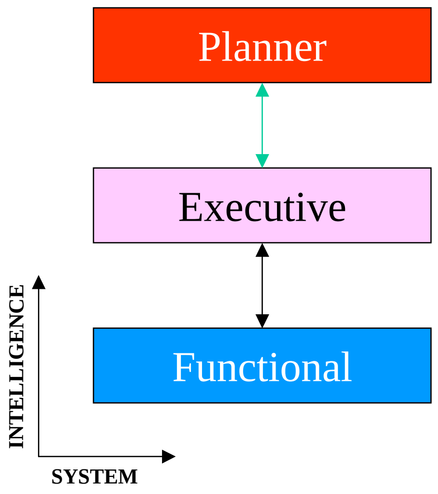


Figure 1.1: Typical three-level architecture.

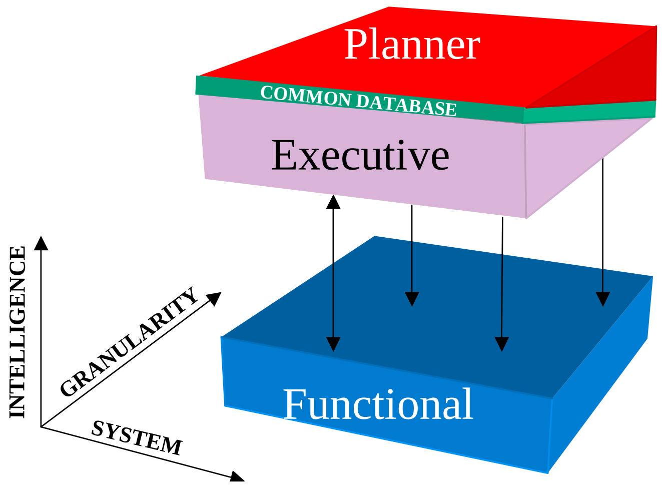


Figure 1.2: Proposed two-layer architecture.

Another problem with this description is lack of access from the Planner to the Functional Level. While this is typically the desirable configuration during execution, it separates the planner from information on system functionality during planning. One consequence is that Planners often carry their own separate models of the system, which may not be directly derived from the Functional Level. This repetition of information storage often leads to inconsistencies between the two.

A third problem with this description is the apparent equivalence of the concepts of increasing intelligence with increasing granularity. In actuality, each part can have its own hierarchy with varying granularity. The Functional Level is comprised of numerous nested subsystems, the executive has several trees of logic to coordinate them, and the planner has several time-lines and planning horizons with different resolution of planning. Therefore, granularity in the system may be misrepresented by this diagram. Worse, it obscures the hierarchy that can exist within each of these system levels.

1.3.2 Proposed Two-Layer Architecture

To correct the shortfalls in the three-level architecture, we propose an evolution to a two-tiered Coupled Layer Autonomous Robot Architecture (CLARAty), illustrated in Figure 1.2. This structure has two major advantages: explicit representation of the system layers' granularity as a third dimension[1], and blending of the declarative and procedural techniques for decision making.

The addition of a granularity dimension allows for explicit representation of the system hierarchies in the Functional Layer, while accounting for the *de facto* nature of planning horizons in the Decision Layer. For the Functional Layer, an object oriented hierarchy describes the system's nested encapsulation of subsystems, and provides basic capabilities at each level of the nesting. For instance, a command to "move" could be directed at a motor, appendage, mobile robot, or team. For the Decision Layer, granularity maps to the activities time-line being created and executed. Due to the nature of the dynamics of the physical system controlled by the Functional Layer, there is a strong correlation between its system granularity and the time-line granularity of the Decision Layer.

The blending of declarative and procedural techniques in the Decision Layer emerges from the trend of Planning and Scheduling systems that have Executive qualities and vice versa [71, 30]. This has been

[1]The convention employed here is to consider lower granularity to mean smaller granule sizes. Examples include individual device drivers in the Functional Layer, or individual basic plan elements in the Decision Layer.

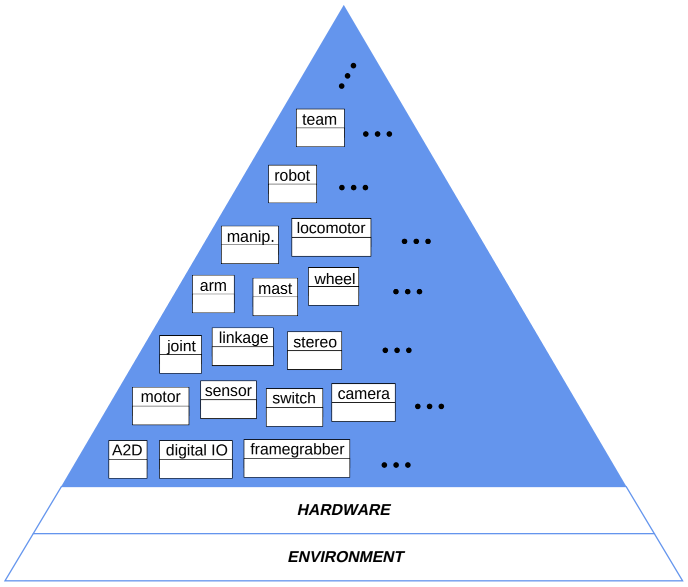


Figure 1.3: Proposed Functional Layer.

afforded by algorithmic and system advances, as well as faster processing. CLARAty enhances this trend by explicitly providing for access of the Functional Layer at higher levels of granularity, thus less frequently, allowing more time for iterative replanning. However, it is still recognized that there is a need for procedural system capabilities in both the Executive interface to the Functional Layer, as well as the infusion of procedural semantics for plan specification and scheduling operations. Therefore, CLARAty has a single database to interface Planning and Executive Functionality, leveraging recent efforts to merge these capabilities [35].

The following sections will develop these concepts by providing an overview of features of both the Functional and Decision Layers, as well as the connectivity between them. Later, in Parts II and III, much greater detail is provided.

1.3.3 The Functional Layer

The Functional Layer is an interface to all system hardware and its capabilities, including nested logical groupings and their resultant capabilities. These capabilities are the interface through which the Decision Layer uses the robotic system. Figure 1.3 shows a very simplified and stylized representation of the Functional Layer. Much greater detail is provided later in Part II. The Functional Layer has the following characteristics:

Object-Oriented: Object-oriented software design is desirable for several reasons. First, it can be structured to directly match the nested modularity of the hardware in a robotic system. Second, at all levels of this nesting, basic functionality and state information of the system components can be encoded and compartmentalized in its logical place. Third, proper structuring of the software can use inheritance properties to manage the complexity of the software development. Finally, this structure can be graphically designed and documented using the UML standard.

Figure 1.4 gives a simplified description of the object hierarchy found in the Functional Layer. In this diagram, a fourth *Abstraction* dimension has been added to illustrate the inheritance structure of the classes in the Functional Layer. At the bottom, a rover object aggregates arm and locomotor objects.

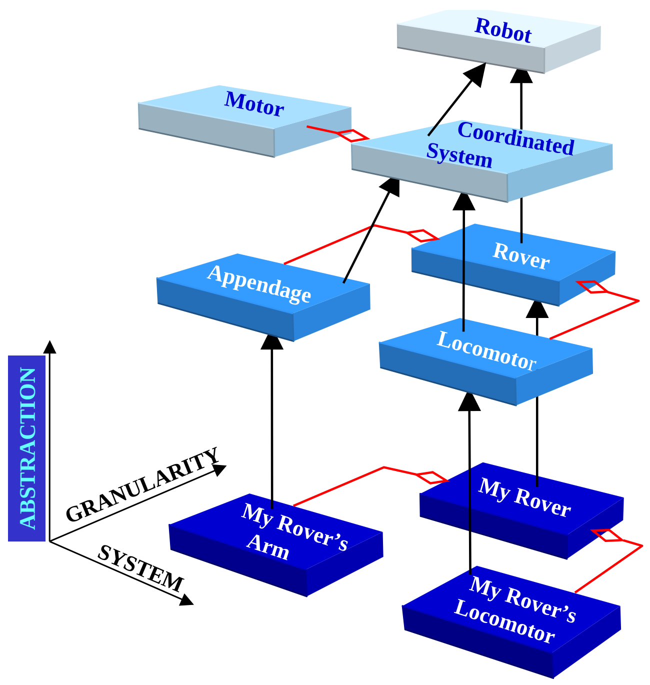


Figure 1.4: Simple example illustrating object hierarchy and Class inheritance concepts.

While these objects comprise a specific *My Rover* system, each is derived from parent classes which are much more general.

An advantage of this structure is that it makes system extension much easier. First, multiple copies of the objects can be instantiated (e.g. two copies of *My Rover's Arm* — left and right). Second, two child classes may inherit all of the Appendage properties (e.g. *My Rover's Arm* and another class, *Your Rover's Arm*, where the latter is somewhat different from the former).

Moving up the class abstraction hierarchy, inheritance relationships may get more complicated. Both Appendage and Locomotor can have a common parent of Coordinated System, which in turn has the same parent as Rover, called Robot. Also, while the Motor class has no children, it is aggregated into the Coordinated System class. In this way, motor functionality is specified centrally in one object and available at all levels below it in the hierarchy, greatly simplifying software maintenance.

Obviously this is just a simple example. Much more detail will be provided in Part II, Appendix A, and subsequent releases of the CLARAty architecture specification.

Encoded Functionality: All objects contain basic functionality for themselves, accessible from within the Functional Layer, as well as directly by the Decision Layer. This functionality expresses the intended and accessible system capabilities. The purpose of this structure is to hide the implementation details of objects from the higher levels of granularity, as well as providing a generic interface.

To the extent possible, baseline functionality is provided in parent classes, and inherited by the children. These children may replace this functionality or add to it. For instance, in the previous example, the *Appendage* class will contain a generic inverse kinematics method, which can be used by its children. However, *My Rover's Arm* may overwrite this functionality with an closed-form algorithm, optimized for its specific design. In addition, it may add functionality specific to the class, such as *stow()* or *unstow()* methods.

In addition to inheritance of functionality, there is also polymorphic expression of functionality. Typically, one member function name is used in all levels of the hierarchy, representing a capability that is appropriate for that level (e.g. *move, read, set, status,* etc.). Since the Decision Layer can access all levels of the Functional Layer hierarchy, it uses this structure to simplify its interactions at different granularity. For instance, a *move* command issued to the *Rover* object would navigate from one place to another using the *Locomotor*, but without a requirement to follow a straight line or find science targets along the way. If the Decision Layer wanted to do the latter, instead of using the *Rover* interface it would access the *move* of the *Locomotor* directly, while also accessing a science target finder object.

Resident State: The state of the system components is contained in the appropriate object and obtained from it by query. This includes state variable values, state machine status, resource usage, health monitoring, etc. In this way, the Decision Layer can obtain estimates of current state or predictions of future state, for use in execution monitoring and planning.

Local Planners: Whereas the Decision Layer has a global planner for optimal decision making, it may utilize local planners that are part of Functional Layer subsystems. For instance, path planners and trajectory planners, can be attached to manipulator and vehicle objects to provide standard capabilities without regard to global optimality. Like all other Functional Layer Infrastructure, the use of such local planners is an option for the Decision Layer.

Resource Usage Predictors: Similar to local planners, resource usage prediction is localized to the objects using the resources. Queries for these predictions are done by the Decision Layer during planning and scheduling, and can be requested at varying levels of fidelity. For instance, the power consumption by the vehicle for a particular traverse can be based on a hard-coded value, an estimate based on previous power usage, or a detailed analysis of the upcoming terrain. The level of fidelity requested will be based on time and resource constraints on the planning stage itself, margins available for the time window under consideration, as well as the availability of more detailed estimate infrastructure. In some cases, subordinate objects may be accessed by superior ones in the process of servicing a detailed prediction.

Simulation: In the simplest form, simulation of the system can be accomplished by providing emulation capability to all the lowest level objects that interact with hardware. In this case, the superior objects have no knowledge of whether they are actually causing real actions from the robot. Such simulation is a baseline capability of the architecture. However, it typically can not be done faster than real-time while using the same level of computer resources. Therefore, it is advantageous to percolate simulation capability up to superior objects in the hierarchy. The cost of this is increasing complexity in the simulation computations. For some purposes such complexity may be valuable. But, as with Resource Estimation, levels of fidelity may be specified to provide useful simulation with reduced computation when desired.

Test and Debug: For initial development and regression testing as system complexity grows, all objects must contain test and debug interfaces and have external exercisers.

1.3.4 The Decision Layer

The Decision Layer breaks down high level goals into smaller objectives, arranges them in time due to known constraints and system state, and accesses the appropriate capabilities of the Functional Layer to achieve them. Figure 1.5 shows a very simplified and stylized representation of the Decision Layer. Much greater detail is provided later in Part III. The Decision Layer has the following characteristics:

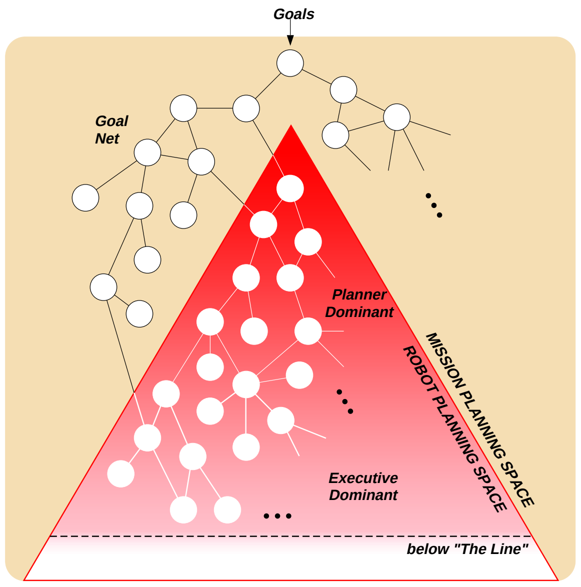


Figure 1.5: Proposed Decision Layer.

Goal Net: The Goal net is the conceptual decomposition of higher level objectives into their constituent parts, within the Decision Layer. It contains the declarative representation of the objectives during planning, the temporal constraint network resulting from scheduling, and possibly a task tree procedural decomposition used during execution.

Goals: Goals are specified as constraints on state over time. As such they can be thought of as bounding the system and specifying *what shouldn't be done*. An example is: 'the joint angle should not exceed 30 degrees or be less than 20 degrees'. Goals may be decomposed into subgoals during elaboration, and arranged in chronological order during scheduling. Resulting goal nets and schedules may be saved, or recalled [27].

Tasks: Tasks are explicitly parallel or sequential activities that are tightly linked. They result from the fixed procedural decomposition of an objective into a sequence, which is possibly conditional in nature. In contrast to Goals, Tasks specify *exactly what should be done* [71]. An example is: 'the joint angle should be 25 degrees'.

Commands: Commands are unidirectional specification of system activity. Typically they provide the interface between the terminating fringes of the goal net, and the capabilities of the Functional Layer. Closed loop control within the Decision Layer is maintained by monitoring status and state of the system as commands are executed [10].

The Line: *The Line* is a conceptual border between Decision-making and Functional execution [27]. It exists at the instantaneous lower border of the elaborated goal net, and moves to different levels of granularity according to the current elaboration. When projected on the Functional Layer, it denotes the border below which the system is a *black box* to the Decision Layer.

State: The state of the Functional Layer is obtained by query. The state of the Decision Layer, which is essentially its plan, the active elaboration, and history of execution, is maintained by this layer. It may be saved, or reloaded, in whole or part.

1.3.5 Layer Connectivity

Given the two architectural layers, Functional and Decision, there is flexibility in the ways in which these may be connected. At one end of the spectrum is a system with a very capable Decision Layer, and with a Functional Layer that provides only basic services. At the other end of the spectrum is a system with a very limited Decision Layer that relies on a very capable Functional Layer to execute robustly given high level commands. If both a capable Decision and Functional Layer are created then there may be redundancy — however, this is seen as a strength of CLARAty, not a weakness. It allows the system user, or the system itself, to consider the trade-offs in operating with the interface between the layers at a lower or higher level of granularity.

At lower granularity the built-in capabilities of the Functional Layer are largely bypassed. This can enable the system to take advantage of globally optimized activity sequencing by the Decision Layer. It also enables the combination of latent functionality in ways that are not provided by aggregation of objects at higher levels of granularity in the Functional Layer. However, it requires that the Decision Layer be aware of all the small details of the system at lower granularity, and have time to process this information. For mission critical operations, it may be worth expending long periods of time to plan ahead for very short sequences of activity. However, this model can not be employed always, since it will force the system to spend a disproportionate amount of time planning, rather than enacting the plans. While the plan may provide optimality during its execution, inclusion of planning time as a cost may force the system be very suboptimal.

To avoid this problem of overburdening the Decision Layer, robust basic capabilities are built into the Functional Layer for all objects in its hierarchies. This allows the interface between the layers to exist at higher granularity. In this case, the Decision Layer need not second guess Functional Layer algorithms, and can also use more limited computing resources. Particularly in situations where resources usage is not near margins, or subsystems are not operating in parallel, it is much more efficient to directly employ the basic encoded functionality. It also directly allows for problem solving at the appropriate level of abstraction of the problem, both for the software and the developers.

1.3.6 Time-line Interaction

The interaction of the two architectural layers, can also be understood by considering the creation and execution of activities on a time-line. Figure 1.6 shows the two layers with the sequence of activation highlighted in green. In the Decision Layer, high level goals are decomposed into subordinate goals until there is some bottom level goal that directly accesses the Functional Layer. During planning and scheduling, this process occurs for queries of resource usage and local plans. If high fidelity information is requested from the Function Layer, such as when resource margins are tight, then the Functional Layer object may also need to access its subordinates to improve the predictions.

The resultant activity list and resource usage is placed on a time-line as shown in Figure 1.7, activities on the top and resource usage on the bottom. Scheduling will optimally order these activities to enable goal achievement while not violating resource constraints. This process, however, must be frozen at some point sufficiently far in the future, so that the schedule is self-consistent at the time it is meant to be executed. Also, the time horizon up to which the planning and scheduling is done is also limited to constrain the problem. Both of these time boundaries are shown in the figure.

Inside the Plan Freeze boundary, it is the responsibility of an executive to initiate actions by accessing the Functional Layer. This process is illustrated in Figure 1.6 by the arrows to the Functional Layer, and the green shading of one portion of the object hierarchy it contains. As the actions take place, resources are consumed, typically in slightly different amounts than predicted. The usage is reported to the Decision Layer, where discrepancies possibly trigger conditional parts of the current plan, and are used to modify the

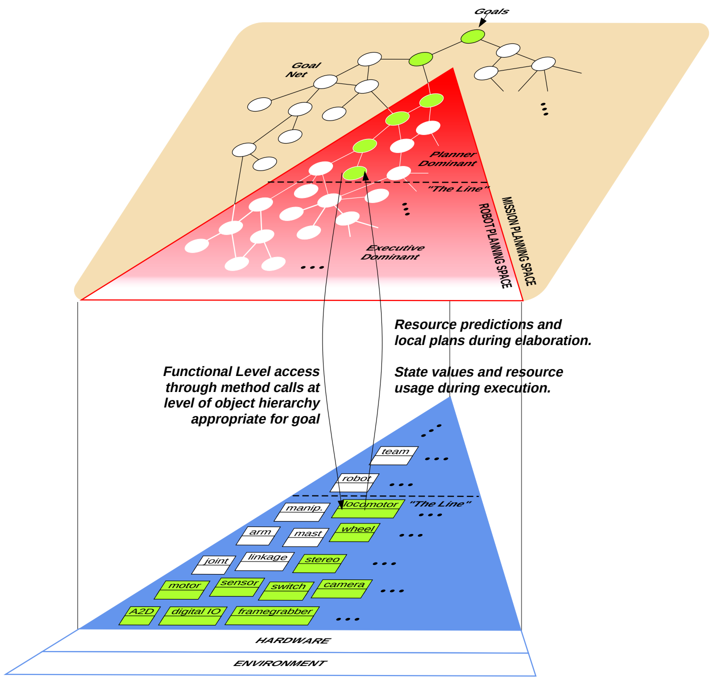


Figure 1.6: Proposed relationship of Functional and Decision Layers.

future projections of resource availability on the time-line which forces replanning to occur. This cycle is indicated by the large arrows in Figure 1.7.

The process described is typical of systems where the procedural components of the executive are separated from the declarative components of planning and scheduling. As will be shown later in Chapter 6, it is not necessary that the boundary between planning and execution exist at a specific point in time — planning and scheduling can occur very near to the present, while executive-style procedural decomposition may be incorporated into distant planning. Therefore, the plan freeze boundary in Figure 1.7 is not required for CLARAty, and the potential cross-coupling of Planner and Executive is one of the primary reasons for merging both into a single Decision Layer. As discussed later, the format of these merged activities, and the interface between them, is currently under development.

Finally, it is important to note that there is also a migration of some executive-style procedural expansion into the Functional Layer as well. Each object has built in functionality which will have a procedural decomposition of its actions, and may have it own mini-executive, or even planner. CLARAty does not preclude this, and allows for this functionality to be leveraged or bypassed, depending on the desire of system designers, and the capabilities of the Decision Layer.

1.4 Implementation

While the prototyping and implementation of the CLARAty architecture is still in its early stages, some specifications and results are important to mention, illustrating the direction of this work. Below are described some of the tool and standard choices, heritage software that will be included into the framework, and prototyping status at this time.

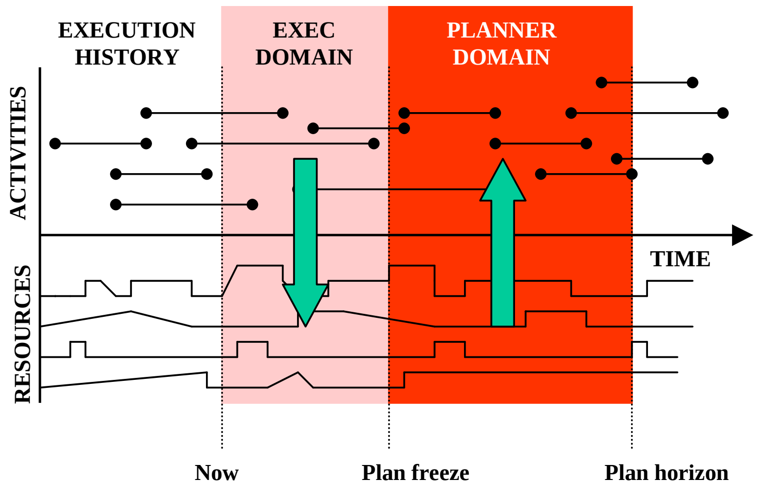


Figure 1.7: Example of system execution time-line.

1.4.1 Tools and Standards

At this point in time, the following tools and standards have been accepted for CLARAty and its development:

The Unified Modeling Language UML is to be used for system design and documentation. The intent is for full use of UML, including templates.

C++ Language: C++ will be used to create CLARAty, due to its wide use in academia and industry, the need for an object-oriented implementation, and the requirements of real-time software implementation.

OS support: To provide both real-time software support while allowing for workstation development, CLARAty will be constructed to run under VxWorks, Linux, and Solaris. Extension to other operating systems in the future is possible.

Standard Libraries: In the spirit of leveraging off public domain standards employed by the software community, software and specifications such as the Standard Template Library, will be employed where possible.

Software Design Tools: While it is possible to build all or parts of CLARAty by writing software directly with a text editor, it is desirable to employ a standard tool for organizing, structuring, and styling the software in a like manner across all developers. Consideration has been given to tools such as $Rhapsody^{TM}$ and $Visio^{TM}$, but no decision is final. Since it is the desire to not prevent wide participation in use of CLARAty, tools with large costs are not desirable.

Documentation: It is important to provide documentation of all components of the system in various forms. The UML was chosen partly for this reason. Other tools for in-line code documentation standardization are being investigated. The intent is to leverage current tools and standards, not to create new ones.

1.4.2 Heritage

While CLARAty is a new architecture design, its design and prototype construction will rely on some important existing infrastructure. First, some of the initial concepts for the Functional Layer object hierarchy were developed by the Planetary Dextrous Manipulators task at JPL [65]. Second, we will use the research rovers *Rocky 7* and *Rocky 8* to frame some of the problems, and as testbeds for prototyped solutions. Third, many years of technology development at JPL and other NASA research facilities have provided valuable software which will be implemented within the CLARAty framework. Among the software slated for inclusion is: JPL stereo vision [85] Carnegie Mellon University and JPL path planning [55, 74], estimation [12], planning and scheduling [23], execution decomposition and monitoring [71], and kinematic and dynamics computing [87].

Part II

The Functional Layer

Chapter 2

Overview of the Functional Layer

2.1 Introduction

2.1.1 Objectives

The main objectives of the Functional Layer are to provide a common platform for robotic research and operation, to provide generic software components that encapsulate well-known robotics functionality, to provide an easy interface to these components, to attach appropriate generic components to different hardware, and to provide a mechanism to extend and modify this framework to accommodate the requirements of various systems.

The Functional Layer provides a flexible platform for the research, development, and integration of new capabilities for robotic systems. Because of the layers of abstraction that the Functional Layer provides, one can carry research at any level ranging from control, sensing, and communication, to vision-based navigation, manipulation, position-estimation, to multi-robot coordination.

The Functional Layer also provides generic solutions to common robotic problems. For example, the manipulation domain provides forward and inverse kinematics for generic manipulators. The vision domain provides two-dimensional image processing and three-dimensional map generation and handling. The mobility domain provides classification and control implementations for various types of mobile robots.

The Functional Layer framework has a rich set of well-integrated components that are simple for non-experts to use, yet flexible for experts to extend and modify. As the knowledge in a particular domain matures, the implementation of that knowledge is moved from the extended framework into the core framework to become accessible to a larger community.

The components of the Functional Layer can be extended, modified or even replaced in order to accommodate various requirements. Each component has some default functionality/behavior, which provides a starting point for using that component. This is useful, in particular, for people who need the basic services of a component that is outside their area of expertise. With a proper design, implementation, and iteration process, the Functional Layer will evolve to accommodate and reflect the needs of many robotics projects.

2.1.2 Challenges

There are several challenges in designing a general Functional Layer that is suitable for several robotic systems. These challenges stem from the variability in the mechanical design, the variability in the electrical design, the variability in the development and operating environment, and the inherent complexity of the inter-disciplinary robotic systems.

Several challenges stem from the large mechanical variability in robotic systems. Robots come in different shapes and sizes and have different capabilities. They vary from the small hand-held rovers to automobile-sized rovers, from tower-like indoor systems to ruggedized outdoor explorers, from tabletop arms to hyper-redundant snakes. The mobility mechanisms of these systems, if any, vary considerably. Some use wheels while others use legs. Some use active suspension while others use passive ones. Further, each robot can have other additions such as arms, masts, and instruments.

Besides all these physical differences, there are electrical hardware differences. For example, one robot might use a PCI backplane for computation and control, another might use a VME backplane, and a third might use serially linked modules. Each hardware component introduces architectural constraints on the system. Consider, for example, the three different implementations of a coordinated motion control system. One might implement the motion control system using commercial off-the-shelf (COTS) motion control boards. A second might use custom designed boards with COTS chips. A third might use a software implementation of the control and coordination algorithms using a host processor. While these are three very different implementations of a motion control system, a person developing vision-based navigation for a mobile robot should not be required to have intimate knowledge of these details; nor should they have a particular implementation inadvertently influence his/her design of vision-based navigation algorithms. In this case, it is more suitable for this person to use an abstract representation for motion control that defines what the component is supposed to do. This component should hide the details of the implementation without compromising particular features of the hardware.

Another source of differences comes from the choice of the operating system and development environment. Different robots use different operating systems. Some operating systems, such as VxWorks, have hard real-time performance characteristics, while others, such as a standard Linux OS, have only soft real-time capabilities. These operating systems also have different models and implementations for running multi-tasking and multi-threaded applications. These differences affect the design and behavior of the system's components. Choice of real-time operating system and development language are also factors that present challenges to developing a generic system. We selected C++ for the development of the object-oriented framework since it balances efficiency and flexibility needed by robotic applications. C++ tools and support for embedded applications have grown in the recent years. Currently, the components in this framework have been targeted to run under VxWorks real-time operating system since the latter has been validated in space flight. However, several of these components also run under both Unix and Linux.

The last challenge comes from the interdisciplinary nature of robotic systems. Each robot has its own level of complexity and adding generic functionality that is not be applicable to that particular system means adding unwanted overhead and complexity. Building monolithic software components that work with all these variations is an impossible task. However, we can design components that can be reused in similar robotic systems. These components must be lightweight, modular, and extendible. The extent of reusable software will vary from one application to another depending on the commonalities among these systems. As the differences between components outweigh the commonalities, the benefits of software reusability start diminishing.

2.2 Approach

To address the challenges presented by the large variability of robotic systems, we use an object-oriented framework that separates abstract functionality from the actual implementation. In this section, we introduce the various concepts used in the design of this framework. Each of these concepts is expanded in subsequent sections.

In Section 2.2.1, we introduce the component-based decomposition of the Functional Layer and present the different types of components. In Section 2.2.2, we present the relationships among these different components, and in Section 2.2.3 we package these components based on the knowledge they provide in the various domain areas. Finally, in Section 2.2.4, we present the organization of these groups of components for particular robotic applications.

2.2.1 A Component-based Decomposition

So, what is the proper decomposition for a generic robotic system? In large, it depends on what elements of the software are targeted for reuse in future applications. One such decomposition can highlight the runtime model of the system, while another can highlight the behavior of the components of the system hiding the runtime models and their implementations. Under different hardware architectures, the runtime implementation of components may change making it desirable for encapsulation. The$ControlShell^{TM}$ software, for example, is a commercial package that highlights the runtime behavior of a system providing a close monitoring its runtime models [68]. Alternatively, our decomposition highlights the behavior of the components of the system while hiding their runtime models and implementation details. Previous designs based on highlighting the behavior of components have been researched, implemented, and tested on several robotic platforms [66, 56, 65].

The behavior of components is usually the invariant element among multiple applications. 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 image depends, largely, on its implementation. In some systems, an analog camera is connected to a framegrabber mounted onto a computational backplane. In other systems, a digital camera is used and the image is transmitted through a fast serial interface directly to the host memory. In either case, the primary function of the imaging system remains the same, i.e. to acquire images. We can represent such a system by an abstract camera component that publishes a uniform interface but hides the details of its implementation and the runtime models. Another similar example is the motion control scenario that was presented in Section 2.1.2.

We will present a classification based on the abstract physical and functional components of the system that we have been evolving over several years. We have used a number of these components in several robotic applications. We employed an object-oriented class decomposition to provide several abstractions for the components of the systems. These components highlight their functional interfaces and encapsulate their implementations and runtime behavior. Abstract components can be extended to have several implementations and runtime models to support different applications. These abstract components attach to hardware components in real systems or to simulation components in virtual systems. Components are implemented using classes. The terms are used interchangeably in this document.

There are three main types of classes in our Functional Layer: (1) data structure classes, (2) generic classes (physical and functional), (3) specialized classes (physical and functional). All three types of classes contain domain knowledge from different disciplines. They are integrated in a framework to maximize code reuse, eliminate duplicated functionality, and simplify code integration. As a result, there are relationships and dependencies among the various classes. Together they provide a modular but well-integrated solution.

Next, we will briefly present these different types of classes. A brief description of the relationships among these components will be presented in Section 2.2.2, with a detailed explanation and examples presented in Section 3.1.

Data Structure Classes

The data structure classes are classes that provide handling, transformation and storage of data. Examples of such classes are: `Vector`, `Matrix`, `Image`, `Message`, `Bit`, `LinkedList`, `Container`, and `String` as shown in Figure 2.1. Further description of the data structure classes and their relationships to one another is presented in Section 3.1.

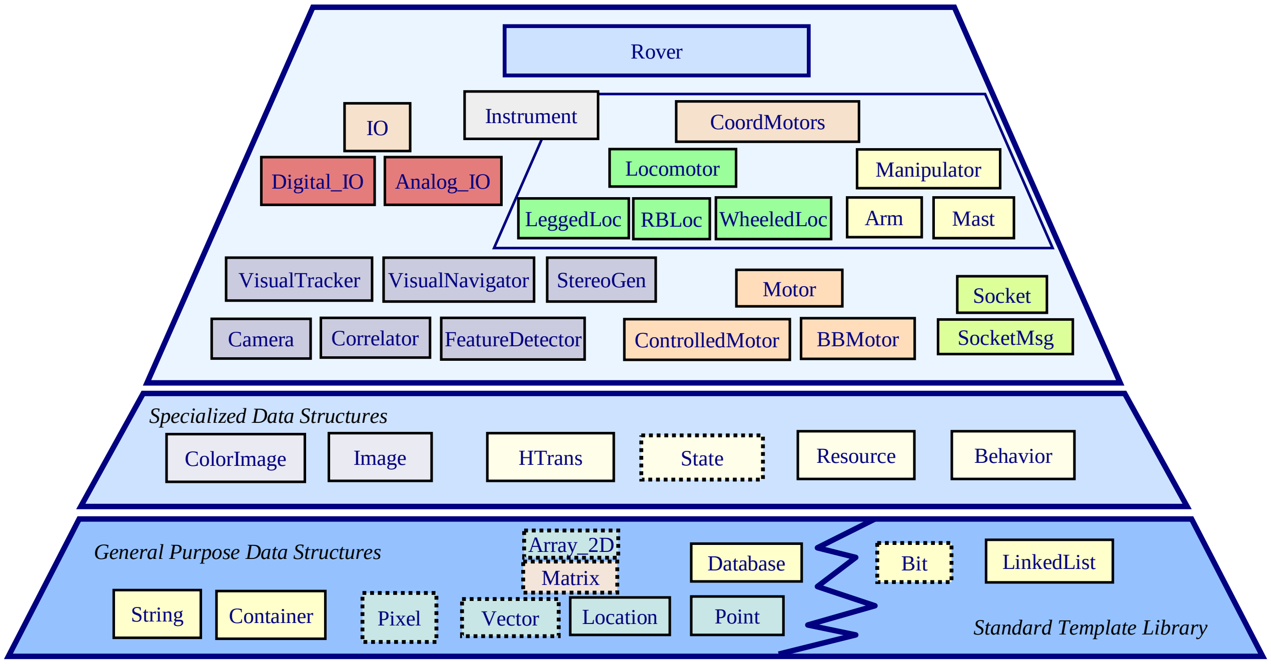


Figure 2.1: Various classes useful for robotics applications.

Generic Physical and Functional Classes

Generic classes are classes that provide an abstract description and implementation of the behavior of a component. These classes can be either generic physical classes (GPC) or generic functional classes (GFC).

A *generic physical* class is an abstract class that describes the properties and behavior of a physical component. These classes expose the capabilities of the components independent of the underlying hardware configuration. Because these components are not directly tied to hardware, they often have partial implementations of the functionality. The extent of the implementation depends on the knowledge available to that class at that particular level of abstraction. Examples of generic physical classes are: `Motor`, `Joint`, `Wheel`, `Arm`, `Mast`, `Locomotor`, `Camera`, `FilterWheel`, `Gyro`, `DigitalIO`, `Socket`, and `SunSensor`.

A generic functional class is an abstract class that describes the interface and functionality of a generic algorithm. A generic functional class can have a complete implementation of its functionality because it interfaces with generic physical classes. Examples of generic functional classes are: `State`, `StereoVision`, `TrajectoryGenerator`, `VisualOdometer`, `ObjectFinder`, `VisualNavigator`, and `Localizer`.

Both types of generic classes can be active, i.e. their objects can generate separate threads of execution and run within multiple threads. In other words, these classes can have local executive capability. For example, a `Motor` class can generate two threads of execution: one for control and the other for feedback. Some also have local planning capabilities.

Specialized Physical and Functional Classes

Specialized classes are extensions of the generic classes that adapt the generic components to the actual physical robot platform. Just like the generic classes, there are two types of these classes: specialized physical classes (SPC) and specialized functional classes (SFC). The specialized nature of these classes makes them suitable for single use only.

A *specialized physical class* is a class that adapts the functionality of a generic class to a particular application. Hence, a specialized class is derived from its generic counterpart. It completes the implementation of its generic parent and in some cases overrides the generic implementation by one that is suited for the particular robotic system. This process is known as the adaptation of the generic framework.

An example of a specialized physical class is found in the Rocky 7 rover implementation. Rocky 7 is a Mars rover prototype that has a three degree-of-freedom mast [79]. During the adaptation process of the mast software, the generic `Manipulator` class is specialized to an `R7Mast` class. The `Manipulator` class provides generic forward and inverse kinematics, joint motion control, trajectory tracking, conditional motion, and error recovery. The specialized `R7Mast` class specifies the links dimensions, joint limits, actuator type, and end effector type. It also overrides the generic kinematics of the `Manipulator` class with the closed-form kinematics that are specifically derived for this type of manipulator.

A *specialized functional class* is a class that is derived from its generic counterpart: the generic functional class. It is only used in cases where an application requires more than parameter adjustments of the algorithms. This specialized adaptation allows the user to modify the functionality of the generic algorithms and override certain operations for a particular implementation. For example, a correlation-based `StereoVision` class can have a specialized feature-based `FeatureBasedStereoVision` class that is less common than its default correlation-based parent. Alternatively, specializated functional classes can be implemented using templates rather than inheritance for a more efficient implementation.

Similar to their generic counterparts, these specialized classes can have executive capabilities. These executive capabilities encapsulate the details of the threading model and implementation that are unique to an existing hardware platform. Such encapsulation enables the design of higher-level abstractions (generic classes) without worrying about system specific details.

2.2.2 Relationships among the Different Components

There are two types of relationships among these components: inheritance, and aggregation [25]. As we have just seen, the relationship between generic and specialized components is that of inheritance. Specialized classes are derived from the generic classes. Both generic and specialized classes are of the same type. In aggregation, however, the aggregated component has a different type than that of the aggregate. Aggregation is used to provide components with different levels of granularity. For example, a `Manipulator` class aggregates lower-level `Motor` class and `Link` class objects.

The reason why such a decomposition of robotic systems is possible is that components at the lower levels of granularity can be implemented with little or no knowledge of their neighboring components. In other words, the coupling among low-level components is loose for the most part. The coupling among these components increases as we move to higher-level components. Higher-level components aggregate lower-level components. Higher-level components manage the interaction of their subordinate components. This approach abstracts the functionality of components and reduces the complexity of the system significantly.

For example, a `Motor` class can be implemented without knowledge of a `Link`, `Manipulator`, `Camera` or `EndEffector` class. Although the `Manipulator` class aggregates the `Motor` class, the `Motor` class does not need to know anything about the `Manipulator` class for the implementation of the `Motor` class. The `Manipulator` class, which aggregates several `Motor` and `Link` objects, is considered a higher-level component. The `Manipulator` class manages and coordinates the interactions between the `Link` and the `Motor` objects.

The above classification works well in systems that have: (a) loose coupling at lower levels and tighter coupling at higher levels; and (b) short execution cycles. In such systems, resources shared by multiple components are handled locally using semaphores. Components rely on the operating system scheduling to handle shared resources and, under proper constraints, appear to operate in parallel. If components use operations that have significant execution profiles, and are tightly coupled to a web of components, we use global resource management to resolve their temporal coupling. This is done by the Decision Layer.

The advantage of this component-level decomposition is that it provides various levels of well-defined abstractions of the system. Additionally, it provides generic and flexible interfaces for developing higher-level components and algorithms. System developers can work at different levels without intimate knowledge of the lower levels. It also simplifies the reconfiguration of various components in the system and their substitution with other implementations of the same component type. For example, if someone develops an algorithm for rover navigation which requires two components: `Locomotor` and `Camera` classes, then this algorithm should work with any type of `Locomotor` and `Camera` objects. Derived types of the `Locomotor` class, such as `WheeledLocomotor` or `LeggedLocomotor`, can be used by the navigation algorithm without the navigation algorithm having knowledge on how these derived classes are implemented or what other classes would be extended in the future. In other words, the navigation algorithm should work on any type of rover whether it uses legs or wheels for its motion, or whether its uses CCD cameras with framegrabbers or uses digital cameras. These details should not appear in the implementation of a generic navigation algorithm. This removes the dependency of the algorithm on a particular hardware implementation.

2.2.3 Packages of Components

A package is a placeholder for a group of components that are closely related to one another. Together these components encapsulate domain knowledge that is applicable to a wide range of systems. Packages localize this knowledge, provide well-defined interfaces, and provide a description of components behaviors and interactions. Users should be able to integrate components from these packages without being domain experts. In this section, we present a brief description of some of these packages.

There are three types of packages. The first type describes software components, the second describes hardware components, and the third adapts the hardware components to the software components. Here is a listing of several packages that contain software components that are reusable across multiple robotic applications:

- The Input/Output package contains classes for bit operations, digital I/O control, and analog I/O control.
- The Motion Control package describes motor-related class hierarchies that include controlled motors, open-loop motors, coordinated motor systems, and trajectory generators.
- The Mobility and Navigation package contains classes that describe various types of locomotors. It also contains a hierarchy of various types of navigation classes.
- The Manipulation package contains various manipulator classes that provide generic solutions to kinematic and some dynamic problems.
- The Vision and Perception package contains classes for two-dimensional image processing algorithms as well as classes for three-dimensional stereo vision and calibration processing.
- The Resource Management package provides tools for local resource management and for supporting resource queries by the Decision Layer.
- The System Control package includes Neural Networks, Fuzzy Logic, Behavior-based Control, and other reasoning sub-packages. Each of these sub-packages provides classes to support the development of algorithms in these domains and their integration with the rest of the packages.

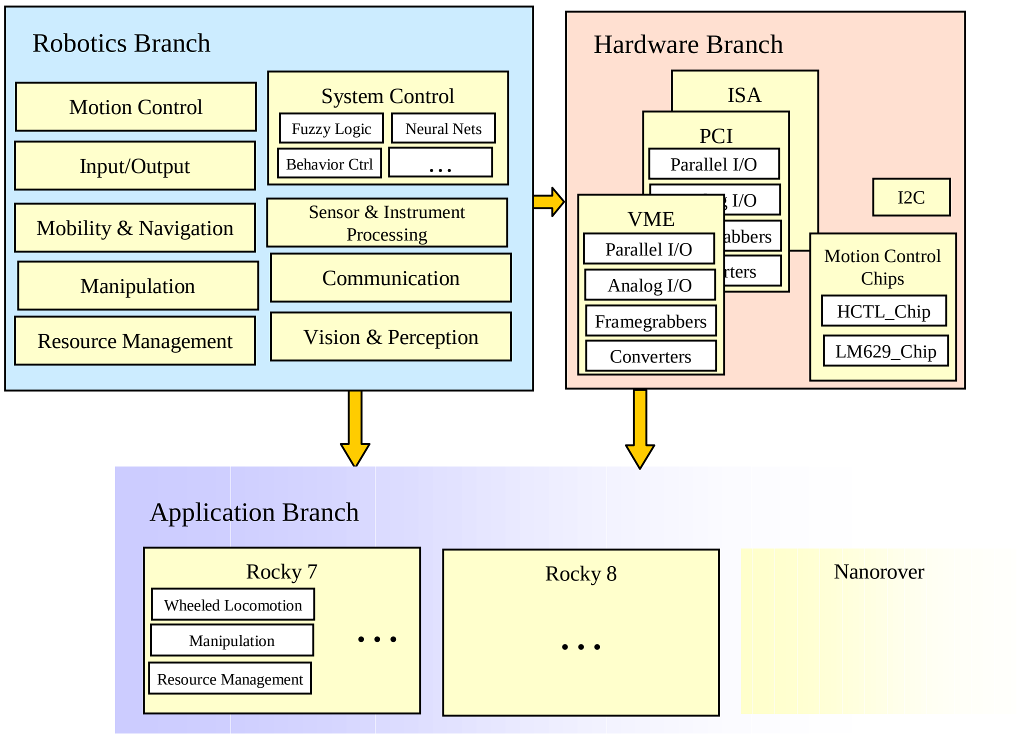


Figure 2.2: Overview of packages and branches in the Functional Layer

- The Communication package provides classes for various serial and parallel communication interfaces. It includes implementations of TCP/UDP socket protocols.
- The Sensor and Instrument Processing package contains generic implementations for various sensing algorithms such as those used for sun sensors.

There can be certain dependencies among these packages. For example, the Mobility and Navigation package can use tools from the System Control package to implement its navigation classes. In addition, to supports legged locomotors, it can use the Manipulation package to implement the legs.

Hardware components are also grouped into packages. These packages are reusable to the extent that the hardware components are reused in different combinations. To make this possible, hardware-related components should provide a complete implementation of the hardware functionality even when the current application does not require such features. Examples of the hardware packages are the VME, PCI, and I2C packages. Each of these packages includes several sub-packages that contain classes representing various I/O boards, motion control boards, framegrabber boards and so on. These classes are tied to software classes through adaptor classes. The adaptor classes complete the implementation of the generic interface and directly ties the generic reusable classes to hardware classes. The adaptor classes are application specific and are also grouped into packages that have similar structure to the reusable software packages, except that they are single use classes and packages.

2.2.4 Branches in the Functional Layer

Figure 2.2 shows the grouping of several packages into branches based on their reusability and hardware dependencies. There are three branches in the Functional Layer: a Robotics branch, a Hardware branch, and an Applications branch.

The Robotics Branch

The Robotics branch consists of a set of packages that contain generic reusable components. The Robotics branch is where the core of the reusable software resides. To date, this branch includes the following packages: (1) Input/Output, (2) Motion Control, (3) Mobility and Navigation, (4) Manipulation, (5) Vision, (6) Resource Management, (7) System Control, (8) Communication, and (9) Sensor and Instrument Processing.

The Hardware Branch

The Hardware branch organizes the different hardware packages. These packages include device drivers for the various boards as well as classes that are shared by boards of the same bus architecture or communication link. The Hardware branch supports different computational backplanes such as VME, PCI, and ISA. This branch also includes support for software implementation of hardware protocols such as I2C bus communication. Other packages include classes that describe the operation of motion control chips, A/D chips, D/A chips, and so on.

Certain classes within the packages of this Hardware branch can use or extend generic classes of the Robotics branch. For example, a particular framegrabber class can use the `Image` data structure from the Vision package in its implementation. It is also possible to have hardware classes specialize generic classes. For example, a particular digital I/O port can inherit the design and interface of the generic port from the Input/Output package.

The Application Branch

The Application branch is a collection of packages that contain specialized physical and functional classes. Its package structure is similar to that of the Robotics branch. Its classes adapt various generic and hardware components to a particular application. For example, a Rocky 7 arm class, called `R7Arm`, specializes the generic `Arm` class of the Manipulation package by specifying the joint and link dimensions, attaching its generic motor to hardware motors, and providing specialized inverse kinematics. Consequently, the classes of the Application branch are typically single-use.

The Application branch can extend classes from both the Robotics and Hardware branches. If classes of the Application branch are useful for multiple platforms, they must be moved to either of the other two branches.

Chapter 3

Components of the Functional Layer

In this chapter, we describe in detail the various types of components (classes) that are used in this framework. We also describe the relationship among these classes. Detailed examples are provided in Chapter 4.

3.1 Data Structure Components

Data structures are the most reused components in the system. There is no single data structure that dominates in the architecture; but there are several types that are used throughout the Functional Layer. The challenge in the design of data structures is to enhance their reusability across the different domains within the Robotics branch. One characteristic of data structures is that they do not have any executive capability, making them the easiest to implement and port to multiple operating systems. While their efficiency is very important, they themselves do not invoke other threads (tasks). However, they must be reentrant to support being simultaneously executed by different threads.

There are two types of data structures relative to our discussion: (1) general-purpose data structures, and (2) domain-specific data structures. General-purpose data structures are reusable beyond the scope of robotics applications. Therefore, whenever suitable, we leverage standardized developments of these general data structures, such as the Standard Template Library implementation [9]. Whenever such implementations are not available for real-time operating systems, or whenever they impose constraints that are not appropriate for robotic applications, we replace them with alternative customized implementations. We maintain the same interface for future replacement.

Examples of general-purpose data structures are: `Array`, `Vector`, `Matrix`, `Bit`, `LinkedList`, `Map`, `Container`, `String`, and so on. Examples of domain specific data structures are: `Image`, `Message`, `Resource`, `Location`, `HTrans` (homogeneous transformation), and so on.

Some domains impose certain constraints on the design and implementation of their data structures. For example, a two-dimensional `Array` class created by instantiating a vector of a vector using the vector class of the *Standard Template Library (STL)* cannot serve as a parent for a `Matrix` class, which in turn is a parent class for the `Image` class. The `Image` and `Matrix` classes must have contiguous memory allocations for their elements for efficient processing. The processing requirements of these two derived classes impose certain constraints on the design of their base class. In other words, a trade-off is made in favor of efficiency over flexibility of the data structure, which influences the design of the `Array` / `Matrix` / `Image` hierarchy.

Next, we present some of these data structures along with their relationship to one another. To date, we have implemented a subset of these data structures that are necessary for robotics applications. We adopt a UML representation to describes these classes and their relationships [25].

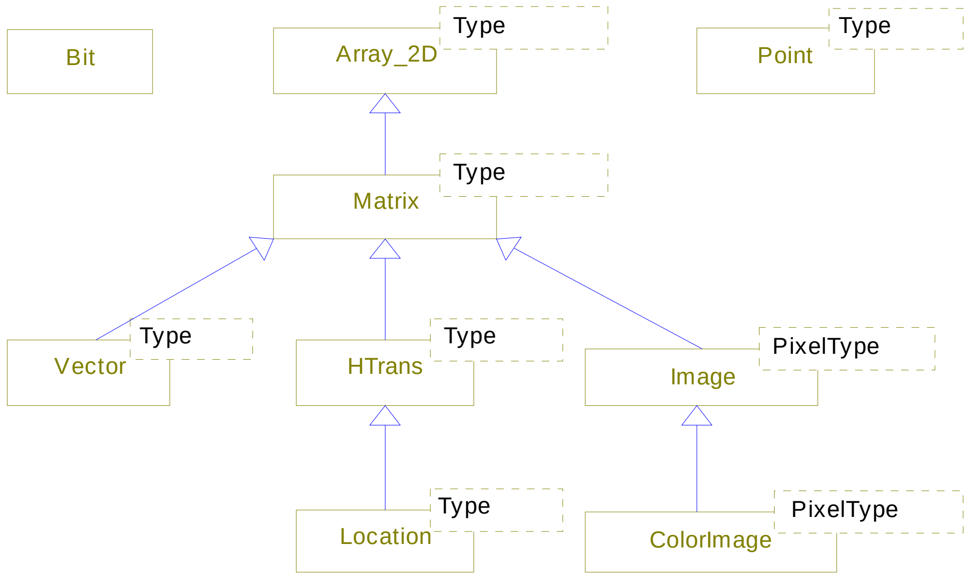


Figure 3.1: Array Hierarchy Data Structure.

3.1.1 The Bit Class

The Bit class manipulates bit patterns of any size. Assignment, shift, and, or, complement, and equality operators are overloaded for this new data type. Additional operations are defined for setting, clearing, resetting, getting, querying, reading, writing, and printing.

Three different implementations of the Bit class have been explored which optimize speed, memory, and a combination of both. One implementation allows for large numbers of bits in an object using a byte to represent 8 bits. An array of 8 bit sets are then created as the core data structure. All the operations for assignment, shift, and equality are overloaded to handle the data type. A second implementation uses one byte to represent every bit. This implementation is not memory efficient since it uses eight times as much memory but it has faster bit shifting and toggling operations. A third implementation allows for bit sequences that are less or equal to the machine limit for bit operations. This class is merely an interface to encapsulate the setting, clearing and counting of bits. By maintaining the same interface, switching between implementations does not change the dependent code.

3.1.2 The Array Class Hierarchy and Image Classes

Figure 3.1 shows the array template hierarchy and the relationships among other classes. At the top of this hierarchy is the Array_2D class, which is a template-based class that implements the creation, deletion, assignment, indexing, row/column manipulations, and resizing of two-dimensional arrays. A Matrix is a template-based class derived from the Array_2D class. The Matrix class defines the mathematical operations of matrices. It defines matrix additions, subtractions, multiplications, inverses, norms, as well as, element-by-element operations in a similar fashion to MATLAB [47]. Averaging, minimum, maximum, equality and many other operations are also defined. A Vector is a template class derived from the Matrix class. A vector is a special type of matrix with a single column. This Vector class defines additional vector operations such as dot product and cross product. An HTrans (homogeneous transformation) is a template class derived from the Matrix class. It describes the position and orientation (pose) of a body in three-dimensional space using a 4x4 matrix. HTrans objects can be concatenated to compute the final pose by computing several intermediate transformations. The Location is a template class derived

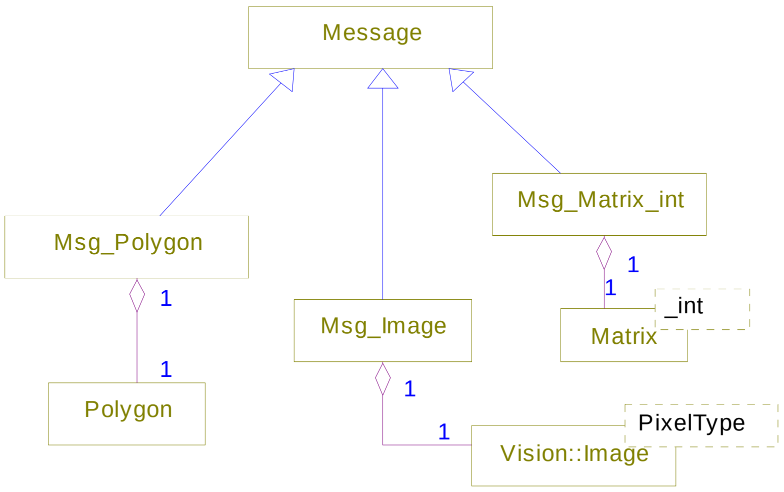


Figure 3.2: The Message class hierarchy.

from the HTrans class. The Location class defines a different interface than the HTrans class. It uses a six-element vector to describe the pose: (x, y, z, pitch, yaw, roll). It uses Euler angle transformations to go from the matrix representation to this vector representation. Although a Point class can be derived from the Vector class, it is more suitable to implement a separate small Point class that can manipulate points very efficiently. Type conversions are implemented so that Point objects can be manipulated with Vector and Matrix objects. This tree will be refined to balance flexibility and efficiency requirements of several applications.

All the above classes are template-based implementations because they have to be useful across several domains within robotics. Consider the Matrix class. If we were to use a floating-point number or double-precision number for the element type, then we could not use the Matrix class as a base for the Image class, which often uses 8 or 16 bit elements for its entries. This is an important consideration since images are usually large and require intensive computational resources. Since our Matrix class is a template class, then, we can derive Image template class from the Matrix class that supports different pixel types. The Image object can be instantiated using any pixel type and size as long as the pixel type supports the necessary mathematical operations defined by the Matrix class. Typically, a pixel type is defined by an unsigned char (8 bit) or short int (16 bit).

A colored image can be represented using three image sets: a red image, a green image, and a blue image, for example. The ColorImage class presenting three colored images is derived from the Image class to maintain the same image type. It also uses the Image class to represent the different color bands.

3.1.3 Message Class Hierarchy

Another class hierarchy pertains to data structures for communication purposes. Figure 3.2 shows the different types of message classes and their relationships to other data structures. At the top of the tree is a generic Message class for transporting large streams of data across a communication link. A Message object has header information that supports breaking up a long stream into a set of small sized Message objects for sending across a communication link. This feature is useful for socket communications using UDP protocols as well as I2C communication of long data streams. The Message class is also used for TCP socket communication. The Message class and its derived types support marshalling and unmarshalling of various class objects, using safe byte ordering for transport to heterogeneous computational platforms. For example, a Msg_Image class is derived from the Message class. It aggregates the Image class. Note that the Msg_Image class is of Message type and not of Image type. The Msg_Image generates the byte stream using the Message header and the Image object. It can also parse the byte stream and store the information back into an Image object. So to transport an Image object to a host, for example, an Image object is wrapped with Msg_Image constructor and then passed down a communication link. To receive an object, the reverse is done. The received Message object is then parsed based on its type (e.g. Msg_Polygon, Msg_Image, etc.). Similar classes are available for Msg_Vector, Msg_Matrix, Msg_Point, Msg_Bit, and so on.

3.1.4 Other Data Structures

Numerous data types remain to be refined, designed and/or integrated into this framework with applicability across several robotics domains. Other examples of specialized data types include a Behavior class hierarchy used in behavior-based robotics, Fuzzy_Set classes for fuzzy-logic control, Neuron and NLinks classes for neural network implementations.

3.2 Generic Physical and Functional Components

As described in Section 2.2.1, there are two types of generic components: generic physical components (GPC) and generic functional components (GFC). In this section, we will describe each type in more detail.

3.2.1 Generic Physical Components

A generic physical component (GPC) is a class that defines the structure and behavior of a physical object in an abstract sense. Some of these classes have partial implementations since they are eventually attached to physical / simulation objects that complete their implementation. The objects to which they attach are of the same type. These classes can be active, i.e. they provide their own threading model. Examples of such components are: Motor, Joint, Wheel, Arm, Mast, Locomotor, Rover, Camera, FilterWheel, Gyro, DigitalIO, AnalogIO, Socket, and SunSensor. These components appear at different levels of granularity in the Functional Layer.

Figure 3.3 shows an illustration of a typical generic physical component. The characteristics of these generic components are:

1. They represent an abstract view of a physical entity.
2. They attach to concrete physical classes of the same type. The physical classes complete the implementation of the generic class interface.
3. They provide generic public interfaces that supports different physical implementations. The interfaces define the functionality and services of the component.
4. They provide the runtime model for component's operation.
5. They manage local atomic resources and resolve local conflicts.
6. They encapsulate the states of a component and provide access to the states through their public interface. The Decision Layer can query any state of a component at any time.
7. They provide local state estimation based on information available within the scope of the component. They may attach to external generic estimators (e.g. Kalman Filter) [20].

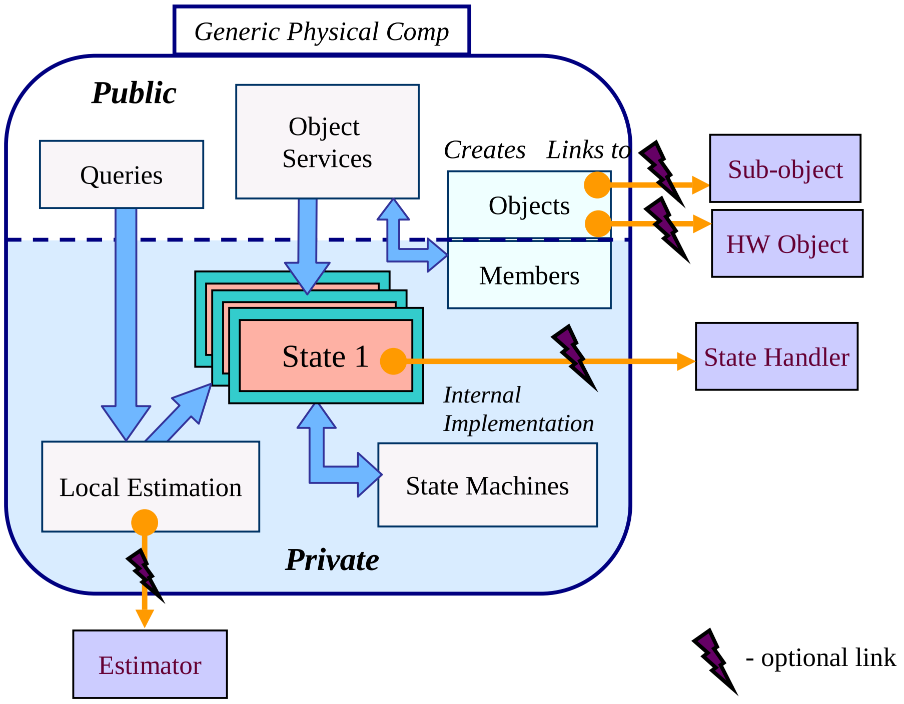


Figure 3.3: A Typical Generic Physical Component (GPC) Structure.

8. They provide resource usage prediction in response to queries from the Decision Layer.

9. They may have internal state machines.

10. They may include or reference other generic physical components. Such components are made publicly accessible to allow access to subordinates.

State and State Handler Classes

Components use state variables for logging, tracking, and recovery strategies. Components can have numerous state variables depending on what states are interesting to a particular application. State information can have different forms. It may be contained in a software variable or a hardware register. To track hardware registers, state variables are created to mirror these registers. Doing so enables tracking and logging of a particular state for planning and recovery purposes. Typical components can have tens of states.

A `State` class is designed to provide a uniform handling of all state variables. The `State` is a template-based class that wraps the actual states variables. State variables can be represented by integers, vectors, matrices, bit patterns, etc. The `State` class tracks transitions, time-tags and logs state history. Internal state machines keep track of current states and allowable state transitions. The `State` class can attach to an external `StateHandler` class, which provides more functionality for all states of the system. The `StateHandler` class provides global operations on states, such as periodic monitoring of any selected subset of the system's states. Such state tracking can be selectively disabled or completely eliminated for applications that do not require this feature.

State information can only be accessed through the state query interface. States can be internally monitored by the component or externally monitored by the `StateHandler`, other components, or by the Decision Layer. A public or private operation of a particular component can create a new internal thread to monitor a state variable and act on state transitions. A single state can be monitored by several components simultaneously (i.e. from several threads of control). To do so successfully, the `State` class implementation must be reentrant.

State Estimation

Like state variables, the state estimation can have different forms. The estimation of the local state is implemented within the scope of the component and may be implemented in software, hardware, or a combination of both. If there is redundancy in the information available to a component, it is used to provide better estimates of the state. While estimation of a state is typically limited to the knowledge available to the component, more sophisticated estimates can be obtained by querying higher-level components that have larger scope. State estimation occurs upon request, either external or internal, at which time the component executes the proper estimation operation, updates the state variable, and returns the estimate.

Resource Queries

In addition to state queries, these components support resource queries. At any time, a component can be queried about the resources required to execute an operation and returns the information to the client. The information can be in the form of a single number, a vector presenting the resource usage profile, or a set of profiles.

Local Execution and Planning

Both generic physical and functional components can have local executive and planning capabilities. While this is limited to the scope of the component, higher-level components enjoy executive control over their subordinates. Global resources, such as power and memory, that couple all components of the system are managed by the Decision Layer. In some sense, the Functional Layer provides different granularity of baseline functionality for the Decision Layer. Higher-level components hide the complexities of their subordinates.

3.2.2 Generic Functional Components

Generic functional components (GFC) are similar in structure to generic physical components except that they do not attach to hardware or simulation components. They provide a framework for implementing complex functional algorithms. Examples of generic functional components are: `ObjectFinder`, `VisualNavigator`, `StereoVision`, and `Localizer`. The `State` class presented above is also an example of a generic functional component.

Generic functional components may sometimes use generic physical components in their implementation. An example of such a class is the `VisualOdometer` class. This class implements an algorithm combining robot motion estimates with visual information to provide accurate position estimates. It use the `Camera` GPC class to acquire successive images and the `Locomotor` GPC class to get a dead-reckoning estimate of the robot's motion. It then combines the information to provide an a refined estimate of the robot's position. Another example of a generic functional component is the `RoverLocalizer` class, which uses stereo vision from the mast of the rover to improve position estimation. This class uses generic `Mast` and `Camera` classes in its implementation.

Similar to generic physical components, generic functional components publish their interfaces and hide their internal implementations. The complexity of these components varies from one type to another. However, they should all provide an easy to use interface for the novice user.

In addition to executive capabilities, certain generic functional components may have local planning capabilities. One such example is the `VisualNavigator` class, which uses vision to plan paths and avoid obstacles. The `VisualNavigator` class uses `Camera` and `StereoVision` classes for image acquisition and three-dimensional map generation respectively. Using this information, it plans a feasible path in its environment. The `VisualNavigator` class has local planning capabilities considering only the knowledge of its aggregated components. If the `VisualNavigator` class is capable of generating multiple paths, the results may be reported to the Decision Layer for a final selection. The Decision Layer has a larger scope than the `VisualNavigator` class and carries out global planning and optimization taking into consideration resource constraints and other goal requirements of the system.

3.3 Specialized (Adaptor) Classes

Specialized classes are extensions of the generic and functional classes. They specialize generic classes to a particular application. This is known as the adaptation process and these specialized classes are also known as adaptor classes. These specialized classes complete the implementation of their generic counterparts and may override some default implementation if necessary. These specialized classes provide an abstraction that ties the generic components to the actual hardware components.

Creation of specialized classes for a particular rover is by far the most difficult and arduous task in bringing up a new rover system. Each hardware component comes with its own architecture and theory of operation. Each generic component also provides its own behavior and theory of operation. Putting the two together without careful design can result in an architectural mismatch and poor system performance. Ideally we would like to leverage the features of the hardware architecture and at the same time fit it ”nicely” into the generic components. This is the job of the specialized classes, which implement the behavior defined by the generic components using the functionality provided by the hardware components. A complete match of functionality cannot always be accomplished. Therefore, these specialized classes must adapt the hardware to the behavior to the extent possible. Detailed examples are given in the next section.

Specialized classes are typically application specific. In some cases, the generic component types and their interfaces are not sufficient for a particular implementation of an algorithm. As a result, an extended version of the generic component can be used instead. Using the extended classes instead of their generic counterparts limits the portability to different robotic platforms. Algorithms that use generic component types in their implementation will operate using any specialized (derived) types.

Chapter 4

Packages of the Functional Layer

In this chapter, we will present several designs for the different packages. Certain components of these packages have been implemented and tested in several applications. The packages that will be presented are mainly those than of the Robotics branch. A few examples are presented to illustrate how these packages are connected to hardware components using specialized classes.

4.1 Input/Output Package

Figure 4.1 shows the Input/Output class hierarchy. At the top level is the IO class, which only provides a simple state machine that verifies whether a request on an I/O link is valid. An I/O link can have several designations: input only, output only, input and output, or undefined. An input request from an output only port will report an error. Such a validation feature can be disabled at the top and be reflected throughout the system. This feature is useful for all types of I/O whether it be digital or analog.

This `IO` class is the base for the `DIO` (digital I/O) and the `AIO` (analog I/O) classes. The `DIO_Port` class is a single digital I/O port representing a contiguous bit pattern. It inherits from both the `IO` class and the `Bit` class of the data structure package. The `DIO_Port` also uses the `Bit` class in its implementation. Two `Bit` objects are used to implement the input and output masks. In addition to supporting the `Bit` class interface, the `DIO_Port` class adds bit masking for I/O operations. The `DIO_Port` class is a generic physical class that can be attached to any hardware port of the same type.

The `DIO` class is a composite class that can represent a single bit pattern (`DIO_Port`) or a combination of bit patterns (multiple `DIO_Port` objects). It is derived from `DIO_Port` class and it also uses the `DIO_Port` class in its implementation. The `DIO` class treats any combination of bit patterns as a single continuous block of bits. The `DIO` class can take a bit pattern as input, break it up into smaller patterns to reflect the underlying hardware setup, and send out these smaller patterns to the different I/O boards. It can also read from multiple I/O boards (with different drivers) and report the concatenated bit pattern.

The `AIO` class also derives from `IO` class. It overloads the redirection operators to read from and write to analog ports. It attaches to hardware analog I/O channels of the same type. Consider the Rocky 7 rover example. For its analog I/O it uses a VME-based VADC20 board from OR Industries. A driver class, VADC20Board was designed for this board. The VADC20 board has several channels. This hardware feature was represented by a Channel class nested inside the VADC20Board class. The nested class has a scope limited to the VADC20Board class. The Channel class is a specialized class that is derived from the generic physical component, the `AIO` class.

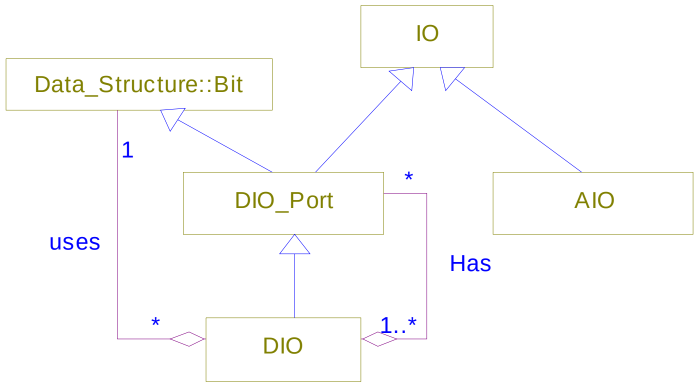


Figure 4.1: The Input/Output class Hierarchy.

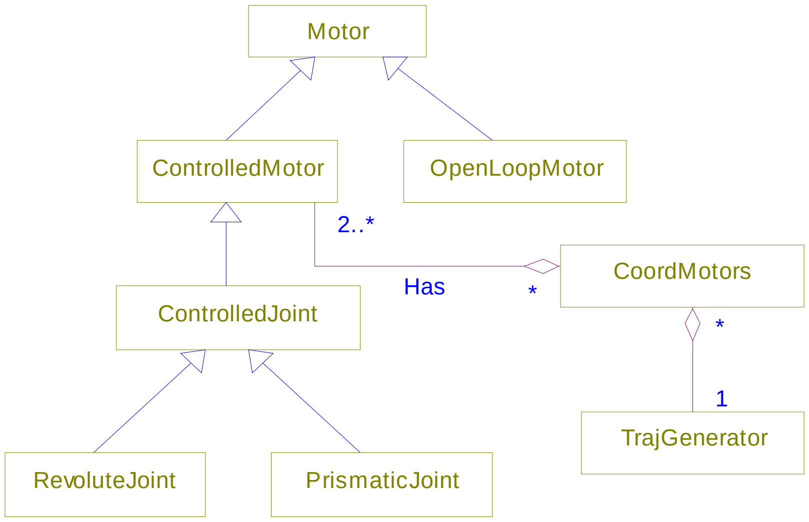


Figure 4.2: The Motion Control GPCs.

4.2 Motion Control Package

4.2.1 Motion Control GPCs

Central to the Motion Control package is the `Motor` GPC hierarchy. Motors are the most common means of actuation in robotic systems. At the top of the tree is the `Motor` class shown in Figure 4.2. This class represents an abstract motor interface common to both closed-loop and open-loop motors. This interface contains operations for enabling, disabling, moving and stopping the motor, as well as queries for their corresponding states. The `Motor` is an abstract class and does not get instantiated but its type can be used for functions that require a very generic motor interface. Both derived types: `ControlledMotor` and `OpenLoopMotor` are concrete classes that can be instantiated.

The `ControlledMotor` class is derived from the `Motor` class and implements a generic interface for closed-loop motors that control their trajectory profiles (position, velocity, and in some cases acceleration). This class is designed to represent servo-controlled motors running different controller chips, stepper-controlled motors, and motors that use customized software-based or hardware-based control algorithms. For applications that do not have motion feedback, the `OpenLoopMotor` class is used. This class shares a

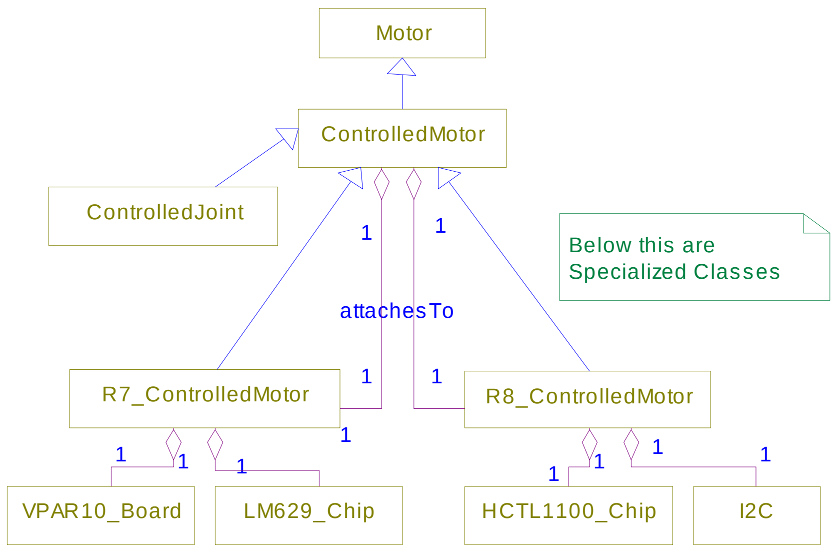


Figure 4.3: A closer view at the functionality of a ControlledMotor class.

similar control interface with the ControlledMotor class but does not have the mechanism for reporting the actual trajectory profile. The OpenLoopMotor class is used for motors driven by a DC voltage level, a pulse-width modulated signal (PWM), or an on-off signal.

The ControlledJoint class is an extension of the ControlledMotor class. It inherits the features of its base class and adds limits on the joint motions. There are two types of controlled joints: (1) a prismatic controlled joint whose interface describes linear motions and constraints, and (2) a revolute controlled joint whose interface describes angular motions and constraints. The two classes representing these joints, PrismaticJoint and RevoluteJoint classes, are derived from the ControlledJoint class. All these classes are examples of generic physical components.

Let us consider the ControlledMotor class whose attributes and interface are partially shown in Figure 4.3 Notice that of the state variables that are tracked and logged are the mode state variable, the _current_position state variable and the _trajectory state vector (includes desired position, velocity and acceleration). The remaining state variables are not explicitly represented in this component, but can be queried through the query interface (e.g. get_real_velocity(), etc.). The ControlledMotor class has a generic interface, which is used for all controlled motors. The interface is divided into three groups: (1) the constructors and initialization functions, (2) the motion and trajectory control functions, and (3) the state and resource query functions. Some of these functions can be implemented using other functions. For example, the get_real_velocity() operation can be computed from the time between successive calls to the get_real_position() operation and the results of these calls. This is useful when this class is attached to an adaptor component whose hardware does not support this feature. In this case, the unavailability of this feature in hardware is overcome through the software implementation.

The ControlledMotor class also implements a generic motor state machine describing the operation modes of the controlled motors as illustrated in Figure 4.4 This class has two parallel state machines when the component is in the On state. Adaptor classes often extend the state machine to include the specialized modes for the particular hardware component. The state diagram shows that the motor can be in a Moving or Not_Moving state at the same time as being 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 or forces from other actuators. The motor can also be in the Servoing and Not_Moving states at the same time, as when the motor stalls. The other combi-

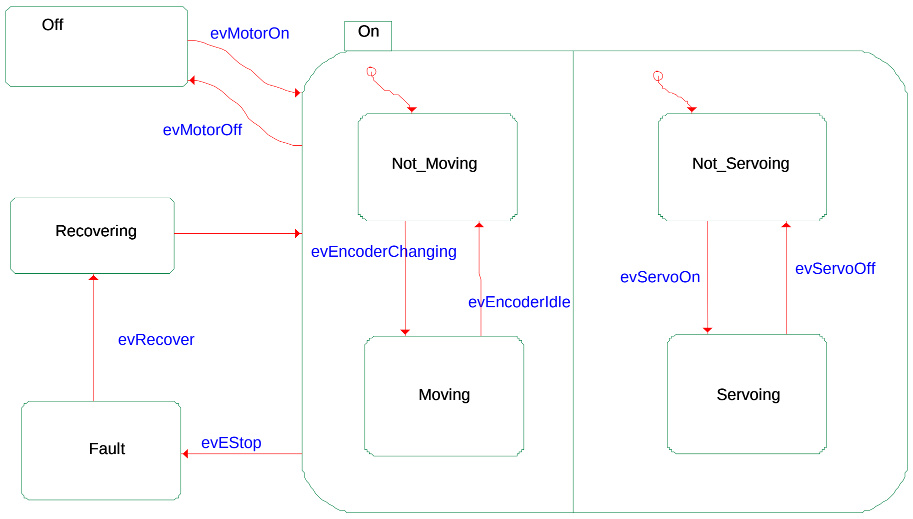


Figure 4.4: The state machine for the ControlledMotor class.

nations are common. From any 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.

The runtime model cannot also be completely implemented in the generic ControlledMotor class, since it varies depending on the configuration of the underlying hardware. However, there are certain runtime features that all motor type classes must meet. First, the operations of ControlledMotor class must be reentrant to support multi-threading on the same object. Figure 4.5 shows two possible runtime models for the ControlledMotor class. 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. In 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 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 ControlledMotor instance for the current position. The ControlledMotor class supports this parallel interaction and manages the communication link that ties it to the physical hardware. Proper protection of local resources and thread management is implemented within the class. Special classes supplied by the resource management domain aid in the implementation of these features. Some operations might also invoke other threads for monitoring and controlling the component.

4.2.2 Motion Control GFCs

Another class related to the Motor class tree is the CoordMotors class (coordinated motors). This class is a generic functional class that coordinates the motion of multiple motors. It does not attach to hardware components directly but rather uses a set of GPC ControlledMotor classes in its implementation. The CoordMotors class requires at least two ControlledMotor type objects to coordinate. It uses several methods for motion coordination. (1) It can attach to a TrajectoryGenerator class (another GFC) and coordinate motion by generating intermediate set points for each motor. (2) It can compute coordinated trajectory profiles (final position, max velocity and acceleration) for all motors and send the results to each motor at the beginning of the trajectory. Each motor will then use its internal set-point generator (hardware or software). (3) It can attach to system models for coordinating motion mainly governed by system dynamics. No matter what method is used, the CoordMotors class is the one responsible for

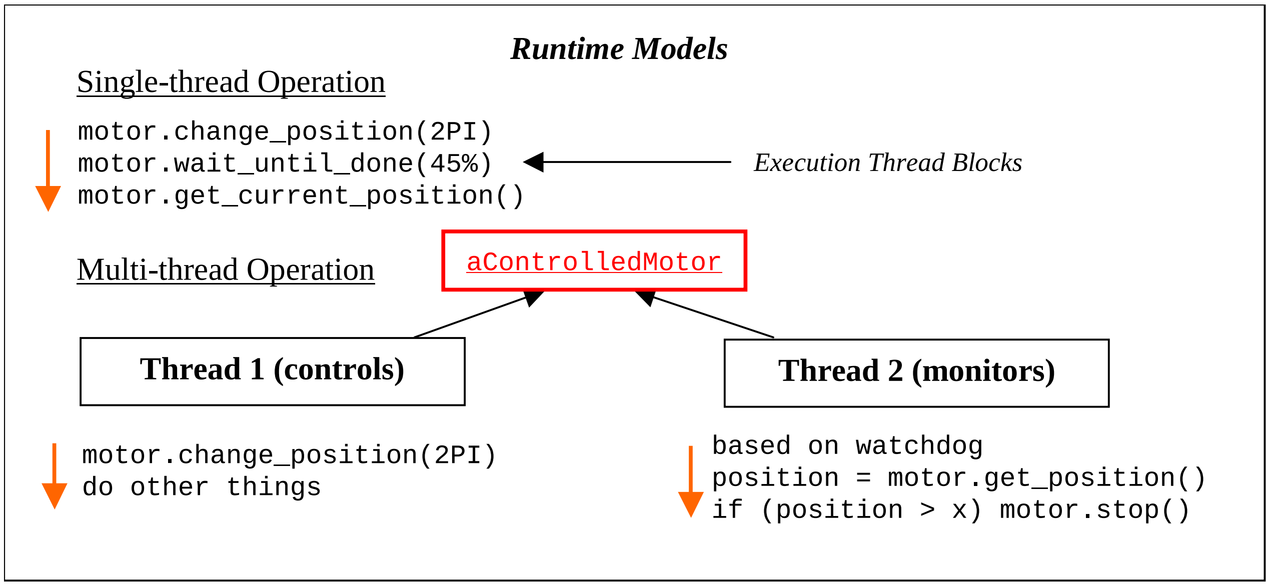


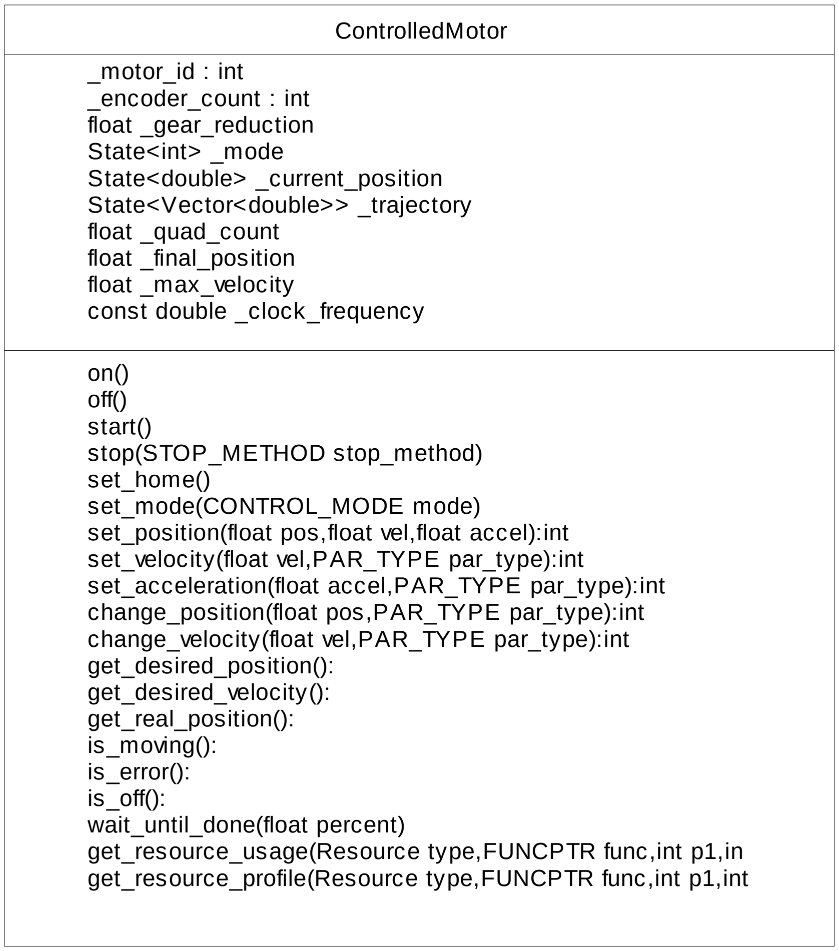
Figure 4.5: Two runtime models for `ControlledMotor` objects.

the smooth coordination of multiple motors or joints. It provides a facade to different internal models for motion coordination. The `CoordMotors` class serves as a base class for higher-level components such as `Manipulator` and `WheeledLocomotor` classes. The `CoordMotors` class is an example of a generic functional component.

4.2.3 Specialized Motion Control Classes

The `ControlledMotor` class provides only a partial implementation of its functionality and behavior as we have just seen. The remaining functionality is implemented inside specialized motor classes that tie the `ControlledMotor` class to classes representing hardware components or to device drivers, which have complete knowledge of the hardware functionality.

This generic `ControlledMotor` class can be used in the implementation of motion control on different systems. Some of these systems can use motion control chips that have local trajectory generators while others can use software to implement motion control and trajectory generation. The behavior of the `ControlledMotor` class after attaching to different software and hardware components should be identical. The specialized classes for these two cases, however, will be different. In the case where the servo loop is implemented in software, several additional tasks (threads of control) must be run at fixed time intervals to execute closed loop control and generate the trajectory profile when a motion command is executed. This runtime threading is encapsulated from the user of the `ControlledMotor` class who is attaching to a software implementation of the servo loops. In the case where the servo loop is done on a motion control chip, these tasks run in hardware loops and the software only sends the motion profile (final positions, max velocities and accelerations) to each motor at the start of the trajectory. While these two systems have different control architectures and executive behavior of their components, it is possible and necessary to abstract their motion control behavior into generic components such as the `ControlledMotor` class. These variations are of no interest to a person trying to design a generic vision-based navigation algorithm for rovers, although the behavior and control of the rover's motion is. As long as the behavior of the generic motion control components is fully characterized, one would know whether the robot supports continuous or discrete trajectory profiling.


```
ControlledMotor

_motor_id : int
_encoder_count : int
float _gear_reduction
State<int> _mode
State<double> _current_position
State<Vector<double>> _trajectory
float _quad_count
float _final_position
float _max_velocity
const double _clock_frequency

on()
off()
start()
stop(STOP_METHOD stop_method)
set_home()
set_mode(CONTROL_MODE mode)
set_position(float pos,float vel,float accel):int
set_velocity(float vel,PAR_TYPE par_type):int
set_acceleration(float accel,PAR_TYPE par_type):int
change_position(float pos,PAR_TYPE par_type):int
change_velocity(float vel,PAR_TYPE par_type):int
get_desired_position():
get_desired_velocity():
get_real_position():
is_moving():
is_error():
is_off():
wait_until_done(float percent)
get_resource_usage(Resource type,FUNCPTR func,int p1,in
get_resource_profile(Resource type,FUNCPTR func,int p1,int
```

Figure 4.6: Examples of `Motor` Specialized Classes for Rocky 7 and Rocky 8.

Consider the Mars rover prototype, Rocky 7 [79]. Rocky 7 has fifteen servo motors controlled using a custom motion control board populated with LM629 motor controller chips. These chips communicate with the host processor, a Motorola 68060 CPU, through a VME-based digital I/O board from OR Industries (VPAR10 board). The communication link between the custom board and the digital I/O is a custom multiplexed shared bus. While this hardware architecture is very unique to the Rocky 7 rover, the software classes controlling LM629, the VPAR10, and the motor are all reusable components. The `LM629_Chip` and `VPAR10` classes are reusable hardware classes, and the motor classes are generic reusable classes. The adaptor classes that integrate all these classes to represent the Rocky 7 motors are specific to this rover and hence not reusable.

For the above scenario, we define a Rocky 7 motor class called `R7_ControlledMotor`. This class is derived from the `ControlledMotor` class as shown in Figure 4.6 It uses the `LM629_Chip` class and the `VPAR10` class in its implementation. The `R7_ControlledMotor` class complements and in some cases overwrites the implementation of the `ControlledMotor` class if an operation is available in hardware and need not use the software version. In other cases, one might choose to combine both hardware and software results for increased robustness.

Since `ControlledMotor` can attach to an `R7_ControlledMotor`, which is of the same type, all derived classes of `ControlledMotor`, such as `ControlledJoint`, can also attach to `R7_Con-`

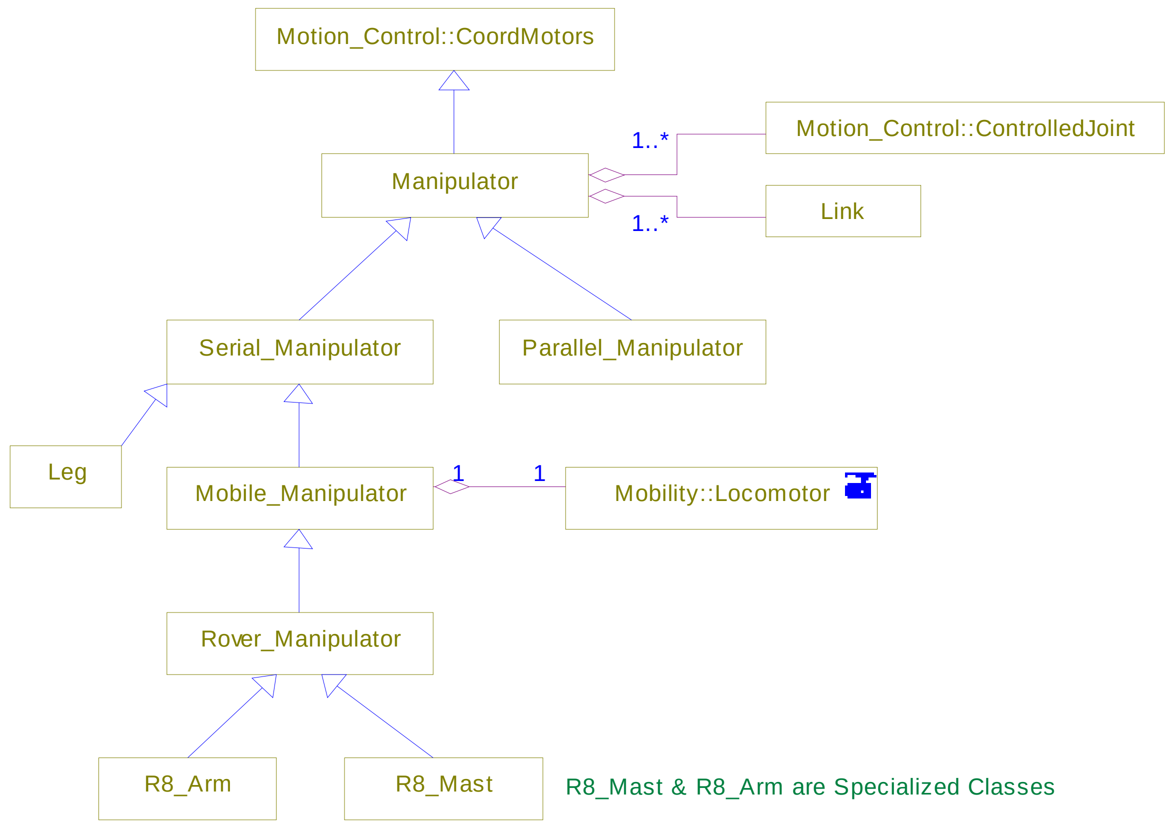


Figure 4.7: The Manipulator class Hierarchy.

trolledMotor. Alternatively, components using ControlledMotor can be passed an R7_ControlledMotor during their instantiation. Either process ties the R7_ControlledMotor to the rest of the system.

The Rocky 8 rover uses a totally different hardware architecture for motion control. The R8_ControlledMotor, which is also derived from ControlledMotor is, therefore, implemented differently. The R8_ControlledMotor uses an HCTL1100_Chip class for motion control and an I2C class for communication with a host processor. However, both specialized classes R7_ControlledMotor and R8_ControlledMotor can be used in algorithms requiring a ControlledMotor data type .

Using this approach, the implementation details and adaptation to hardware are encapsulated from the user. Someone studying multi-rover formations using heterogeneous rovers does not need to be concerned with the differences in implementation of the motion control architecture of each rover.

4.3 Manipulation Package

4.3.1 Manipulation GPCs

This package contains domain knowledge and generic implementations for manipulation classes. Figure 4.7 shows the manipulation hierarchy. At the top of this hierarchy is the Manipulator class which is a generic physical component. This class is derived from the CoordMotors class of the Motion Control package described in Section 4.2.2. It also aggregates an unspecified number of ControlledJoint and Link objects. In other words, a manipulator is a system of coordinated motors that have a number of links and joints. The joints can be either revolute or prismatic. The `Manipulator` class provides functionality such as individual joint mode control and global velocity/acceleration control. It also contains strategies for recovery from error conditions. Additionally, it provides hooks for attaching of various end effectors.

A `Manipulator` object can be queried for the state of several motion variables. Chief among these are the motion state of its joints, the current position, the destination position, the current joint angles, the destination joint angles, and the control mode. A `Manipulator` object can also be queried by the Decision Layer for the resources that are needed for executing a particular command. The `Manipulator` class provides the mechanism necessary to execute the query command and partially implements the request. The rest of the information processing occurs at the specialized manipulator class level.

Two manipulator types can be derived from the `Manipulator` class: the `Serial_Manipulator` class and the `Parallel_Manipulator` class. A serial manipulator is a robotic arm that concatenates a number of joints and links. Industrial robotic arms used in assembly and car painting are examples of serial manipulators. A parallel manipulator is a mechanism whose links are attached in parallel to an output plane. An example of a parallel manipulator is the Stewart platform that is used in motion simulators. There is a duality in the equations governing the kinematics of serial and parallel manipulators. Serial manipulators have relatively simple forward kinematics while parallel manipulators have relatively simple inverse kinematics. On the other hand, closed-form inverse kinematics for serial manipulator and the forward kinematics of a parallel manipulator do not have simple generic implementations[1]. Hence, these kinematic solutions are not implemented in the generic `Serial_Manipulator` and `Parallel_Manipulator` classes. Their implementations are deferred to the specialized physical classes representing the specific manipulators.

For example, the `Serial_Manipulator` class has generic forward kinematic equations that will apply to all types of serial manipulators. However, the closed-form inverse kinematics for serial manipulators are deferred to the specialized classes that are derived from the generic `Serial_Manipulator` class, such as the `R8Arm` class. There are also hybrid manipulators that combine both serial and parallel linkages in their design. We will focus more on serial manipulators since they are more commonly used in robotics applications.

A serial manipulator can be used as an arm or a leg for a robot. It can be mounted on a fixed platform or on a mobile robot. Each of these options requires additional functionality and behavior that a serial manipulator must support. For example, it might be helpful for a manipulator mounted on a mobile platform to know about the mobility system and be able to control it in some cases. One such case is when you are tele-operating this arm. If the arm was not aware of the mobility system, as you extend the arm to the edge of its workspace, the arm looses dexterity and soon becomes singular. But because the arm knows that it is mounted on a mobile platform, then the arm can command the mobility system to advance the robot slightly so as to shift the workspace of the arm forward, keeping the arm in the most dexterous region of its workspace. The arm interface remains the same but its functionality and workspace are extended.

This functionality can be implemented within a `Mobile_Manipulator` class, which uses a generic `Locomotor` class in its implementation. The `Mobile_Manipulator` is derived from the `Serial_Manipulator` class. One type of mobile manipulator is the `Rover_Manipulator` class. In addition to supporting the functionality of a mobile manipulator, the `Rover_Manipulator` class extends the interface of the `Mobile_Manipulator` class to include additional operations, such as `stow()`, `unstow()` and other rover specific functionality.

[1]However, there are some iterative methods that implement generic inverse kinematics for serial manipulators and forward kinematics for parallel manipulators. These are usually computationally intensive.

4.3.2 Specialized Manipulation Classes

Consider the Rocky 8 rover, which defines two specialized classes derived from the `Rover_Manipulator` class. They are the `R8Mast` class and the `R8Arm` class. These classes define the joint configuration and parameters, link types and dimensions, inverse kinematics, and other properties unique to these manipulators. For example, the `R8Mast` class adds some provisions for imaging by using the generic `Camera` class.

4.3.3 Manipulation GFCs

One example of a manipulation GFCs is the `VisualManipulator` class. This class uses the `Camera` and `Manipulator` GPC classes to provide visually guided manipulation capabilities. The implementation of these classes require specialized domain expertise.

4.4 Mobility and Navigation Package

This package describes the various types of mobility platforms that make up the `Locomotor` GPC tree as well as the different navigation classes that represent the generic functional aspect of this package.

4.4.1 Mobility GPCs

Mobility describes the means by which a robot moves. At the top of the mobility tree shown in Figure 4.8 is the abstract notion of a `Locomotor` class. Mobile robots can have different types of locomotion mechanisms. Some mobile robots use wheels, some use legs, while others might hop. For now, we derive two classes from the `Locomotor` class: the `WheeledLocomotor` and the `LeggedLocomotor` classes. The `WheeledLocomotor` class generates motion of the robot by coordinating the trajectory of its wheels and steering joints. These wheels and steering joints are driven by controlled motors. Earlier we have discussed a class that coordinates controlled motors called the `CoordMotors` class. In addition to inheriting from the `Locomotor` class, the `WheeledLocomotor` class also inherits from the `CoordMotors` class.

A `LeggedLocomotor` class uses legs in its implementation. A leg can be implemented as a type of serial manipulator as has been presented in the Manipulation package. A Leg class is derived from a `Serial_Manipulator` class, which, in turn, is derived from a `Manipulator` class (Figure 4.7). While the `LeggedLocomotor` class requires coordinating the motion of its legs, it is a not a coordinated motor system in the same sense as the `WheeledLocomotor` and the `Manipulator` classes are.

4.4.2 Examples of Local Executive Behavior

These locomotor classes are generic physical classes. Some are active components, i.e. their objects can generate separate threads of execution and run within multiple threads. In other words, these classes can have local executive capability. An example of this executive behavior can be found in the `WheeledLocomotor` class. One service (operation) of the `WheeledLocomotor` class is the `calibratesteering()` operation. When this calibration function is called, the `WheeledLocomotor` class creates a calibration thread (task) for each steering unit that needs calibration, so that they all occur in parallel. The actual calibration function for each steering motor is not implemented at this level of abstraction. It is deferred to the specialized classes of `WheeledLocomotor`. However, what is implemented at the `WheeledLocomotor` class is the mechanism that ensures that the object remains alive (does not go out of scope) until all the calibration threads have completed. This is an illustration of a runtime threading model for the steering calibration process specific to wheeled vehicles.

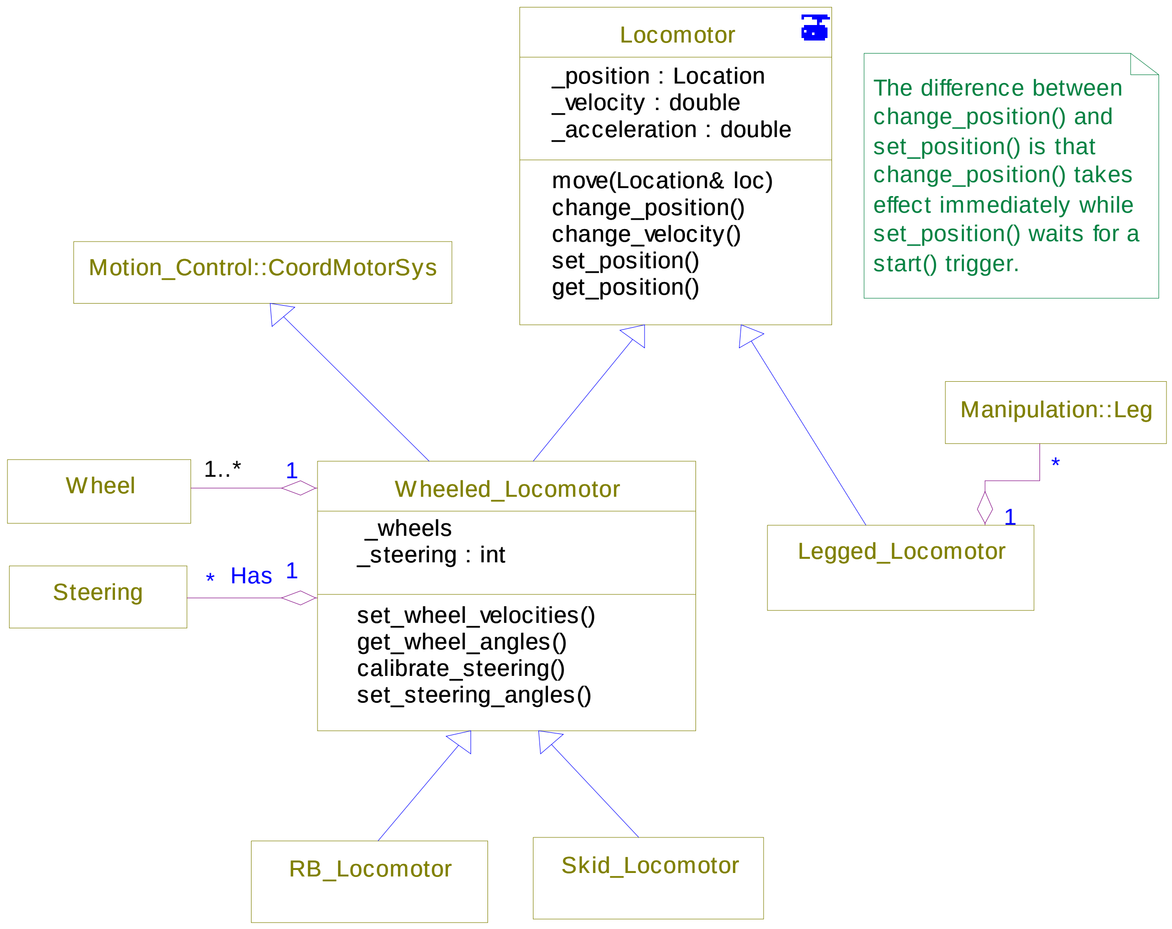


Figure 4.8: A partial view of the Locomotor abstraction tree (mobility domain).

We can take this functionality one step further. The calibratesteering() operation can be encapsulated inside the move() operation of the WheeledLocomotor class. This move() operation overloads the Locomotor 's move() operation and adds some partial implementation that, first, checks if the steering units need calibration, and if so calibrates them. This implementation hides the detailed differences between the generic Locomotor and a WheeledLocomotor . This hidden functionality is still accounted for in the resource predictions given to the Decision Layer. This example illustrates executive capability based on a conditional state of a component where the entire process is hidden behind an abstract interface of a generic class, the Locomotor 's move() operation.

4.4.3 Specialized Mobility Classes

Classes such as WheeledLocomotor and Locomotor only have partial implementations of their functionality. The extent of the implementation depends on the knowledge available to that class at that particular level of abstraction. However, some generic classes have full implementation of their functionality as is the case for the rocker-bogie locomotor class called RBLocomotor. This class implements the kinematics of a six-wheel rockie-bogie suspension mechanism common to many recent Mars rover designs. The behavior of this mechanism is well-defined and closed-form solutions can be obtained. The only specialization of this class is the specification of the mechanism parameters such as link dimensions. The actual parameters for a

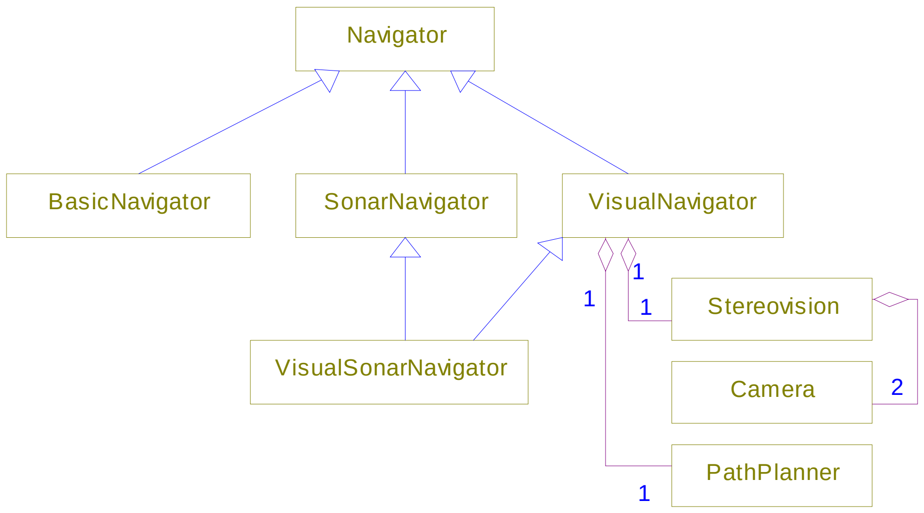


Figure 4.9: The Navigator class hierarchy

particular rover, in this case, are introduced during the instantiation of the actual object.

4.4.4 Navigation GFCs

Some of these classes, such as those related to the navigation aspect of this package, have local planning capabilities. Consider a generic Navigator class for a mobile robot as shown in Figure 4.9. This class provides a type and generic interface for the various types of navigation algorithms. Derived from this class is a VisualNavigator class, which implements a navigation algorithm based on visual information received from its cameras and knowledge about its mobility system. The VisualNavigator class uses the Locomotor class and some vision-based classes such as StereoVision and Camera classes in its implementation. This class also has a PathPlanner class, which generates several possible paths for navigation. While the Navigator class provides some default path selection based on the knowledge available to the Navigator class and its subordinates, a better path selection can be accomplished by the Decision Layer using global planning. Other derived types from the Navigator class are BasicNavigator, SonarNavigator, and SonarVisionNavigator classes.

4.5 Perception and Vision Package

In this section, we will first present some vision-related data structures that are specific to this package. We will then present a design that illustrates how vision-based GPCs can be built. Finally, we will present a few examples of some vision-based GFCs and how they relate to other vision-based components.

4.5.1 Vision-Related Data Structures

Figure 4.10 shows a design for specialized data structures for vision and perception applications. These data structures are centered around the Image class which we have briefly presented in Section 3.1.2. As we have already discussed, the Image class is a template-based class that is derived from the Matrix class. The Image class is parameterized by the pixel type. It is important to keep the Image class small and efficient. Hence, the basic image processing functions are not made part of the Image class.

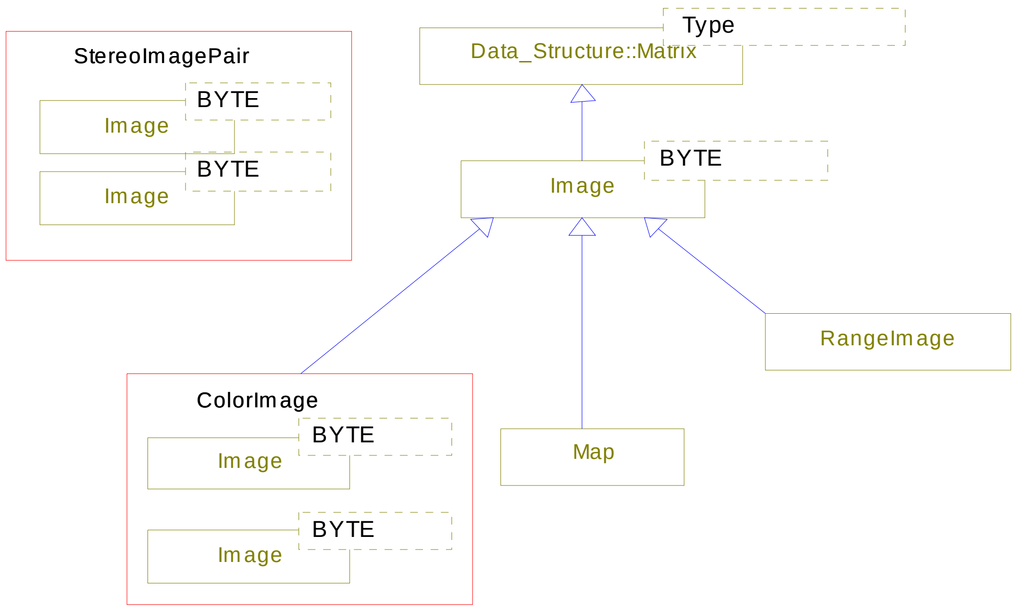


Figure 4.10: The `Image` Class Hierarchy.

Derived from the `Image` class are specialized types of `Image` such as `Map`, `RangeImage` and `ColorImage` classes. A colored image in this tree is represented using three image frames: a red frame, a green frame, and a blue frame. The `ColorImage` class, which is derived from the `Image` class, inherits the array data from the `Image` class to represent the red frame. The green and blue frames are created by aggregating two additional `Image` objects. This design makes the `ColorImage` class maintain its `Image` type to be used interchangeably with the Image class. Alternatively, the StereoImagePair class is not of `Image` type. It contains two `Image` objects, but has its own type. The Map class, which shares a similar representation as an image, supports operations for map manipulation and concatenation. These classes are used to support GPCs and GFCs for 2D and 2 1/2D vision applications.

4.5.2 Imaging and Perception GPCs

Figure 4.11 shows some of the generic physical components that are needed in perception such as `Camera` and `Sonar` classes. The `Camera` class represents, in the most abstract sense, a complete imaging system capable of acquiring images. Since there are several implementations for image acquisition systems, it is important to abstract these to the most general form. In one implementation, an imaging systems may use a camera connected to a framegrabber through an RS-170 analog video signal. The framegrabber may reside in a computational backplane such as cPCI or VME to digitize the analog video signal. The data is then transmitted through the backplane to the host memory. A second implementation may use a digital camera with a direct Universal Serial Bus (USB) connection to the host processor and memory. While these two systems are completely different from a hardware architectural standpoint, they can still be represented using the same abstract `Camera` class with a uniform interface for image acquisition and frame synchronization. These systems will however have different capabilities and performance. Part of the performance will depend on the adaptation to a particular hardware and the other part will depend on the hardware capability. Other related classes in this domain are `Framegrabber`, `VideoSwitcher`, `Camera_FG`, and `Camera_USB` classes.

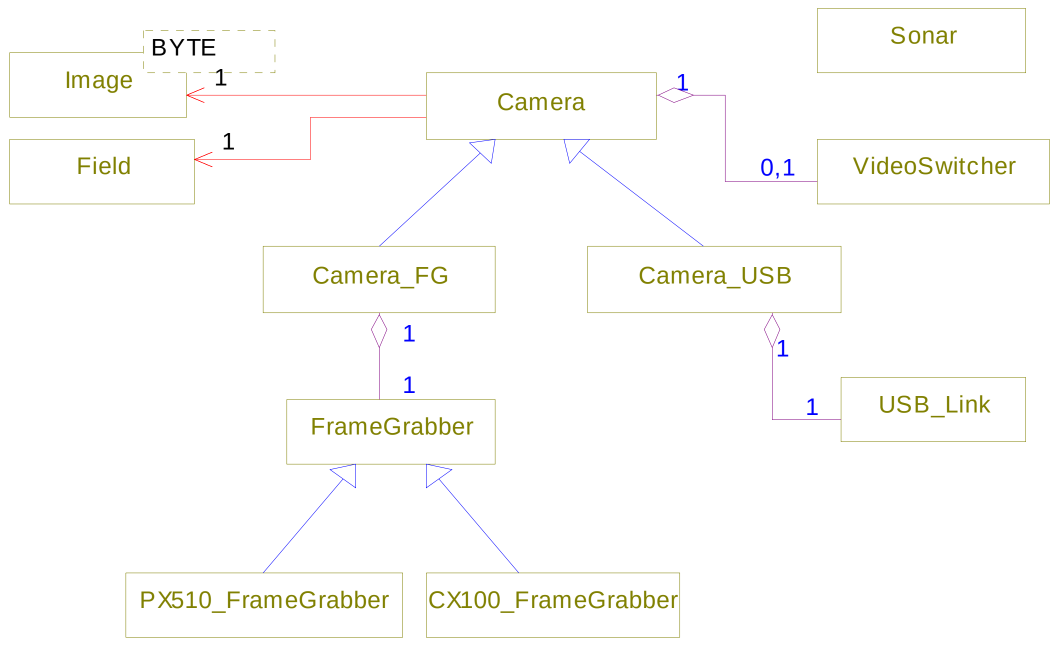


Figure 4.11: The `Camera` class hierarchy.

4.5.3 Vision and Perception GFCs

Image processing functions are not supported directly in the `Image` class. Figure 4.12 an example of some image processing classes and their relationship with other vision-related classes. An `ImageProcessor` class is designed to process `Image` objects. That is how two-dimensional processing algorithms are defined and applied to `Image` objects. Various types of image processors are derived from this generic `Image-Processor` class. For example, different edge detectors are grouped under an `EdgeDetector` class, which is derived from an `ImageProcessor` class. These classes are generic functional classes.

Some of the basic two-dimensional vision processing functions to be supported in this domain are:

- thresholding and histograms
- various types of edge detectors
- various filtering algorithms, e.g. Laplacian of the Gaussian
- frequency domain analysis, e.g., Fast Fourier transforms, Wavelet transforms
- variable-size correlators
- recursive blob analysis and blob statistics
- various geometrical edge and ruler tools that detect transitions and measure image features
- template-matching
- image compression algorithms

For three-dimensional processing, a set of stereo vision classes will support the following growing list of functions:

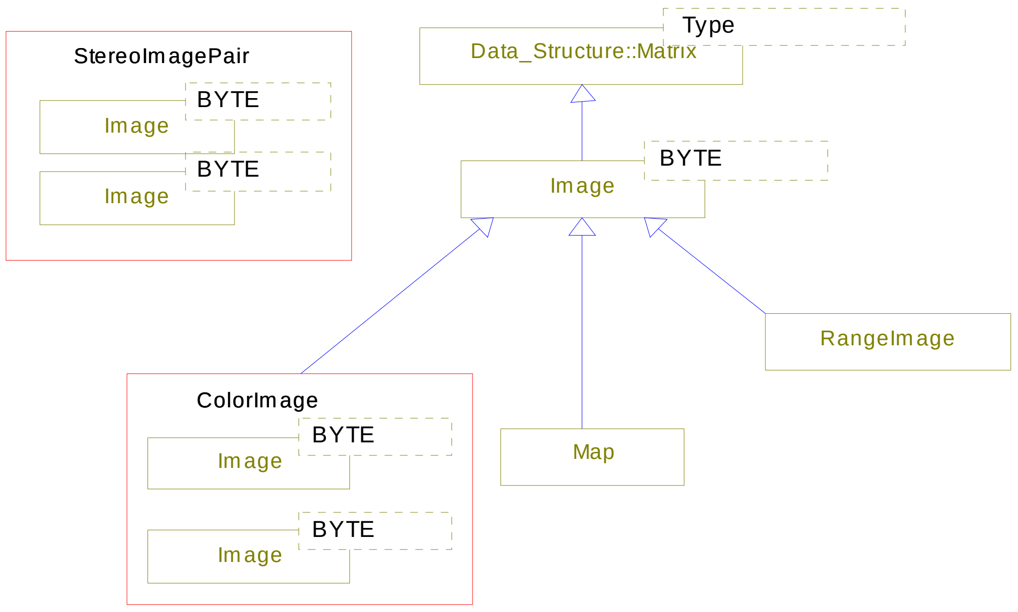


Figure 4.12: Some image processor classes and their relationships with the `Image` class.

- calibration of cameras
- auto-correction of calibration parameter
- generation of range maps
- generation of elevation maps
- elevation map matching (supported in the Map class)
- three-dimensional feature detection

There are several efforts that have attempted to standardize image processing libraries. Examples of these are the Vector Signal Image Processing Library which was initially developed using DARPA funding [3], the ImageVision Library from Silicon Graphics Inc. [1], and the recently announced effort sponsored by Intel to develop a standard image processing library [49]. We will follow these developments in the hope of leveraging off these major efforts. Our main challenge here is to have a package that can be well-integrated with the remaining packages of the Functional Layer and have it be resource efficient to operate on various robotic platforms.

4.6 Communication Package

To date we have implemented two types of communication components: I2C and `Socket` classes. The socket communication classes we have developed are shown in Figure 4.13

The classes we have built run under both VxWorks and Unix systems. At the top of the tree is a generic `Socket` class that initializes a port and selects the type of protocol to be used (UDP or TCP). Two classes are derived from `Socket`: a `Client_Socket` class and a `Server_Socket` class. The `Client_Socket` class can send message of variable length to server socket using either protocol. Both classes use the `Message` class and its children (e.g. `Msg_Image`, `Msg_Matrix`) to transport messages.

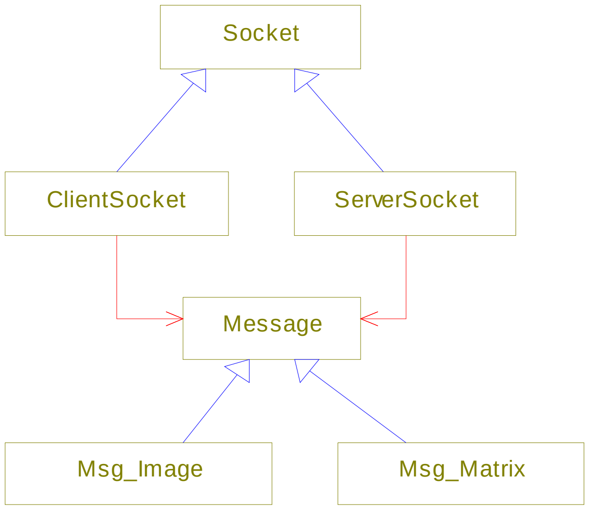


Figure 4.13: The `Socket` class hierarchy.

The `Server_Socket` class listens to incoming messages and responds to multiple clients by spawning a task to service each request. The `Server_Socket` can be operated in either a single message receive mode or in a continuous message receive mode.

The I2C communication class implements a channel for I2C communication. An implementation for I2C master is available for VxWorks.

4.7 Resource Management Package

The Resource Management package contains classes that deal with system resources, both software and hardware. These classes can also be used to abstract the dependencies of components on the operating system calls. For example, the `Task` class handles the operating system dependent task management functions such as creation, deletion, registration and coordination of tasks.

4.7.1 Resource GPCs

Since each component of a system is the most knowledgeable about its own behavior, it follows that each component must have the most knowledge about its own resource usage. Consequently, the Decision Layer queries the Functional Layer components for resource usage predictions.

As shown in Figure 4.14 there are four types of resources: (1) depletable resources, (2) non-depletable resources, (3) concurrent resources, and (4) atomic resources. A depletable resource is a resource that gets depleted with usage and time such as a non-rechargeable battery. A non-depletable resource is one that can be replenished forever such as memory or solar cells. Atomic resources are resources that are either available or busy such as a digital I/O line.

4.7.2 Queries for Resource Usage

The Decision Layer queries the Functional Layer for resource predictions. This is not limited to power request. Other resource requests include time estimates for executing a particular operation, memory usage, and so forth. Some resources are more significant for particular components. For example, a

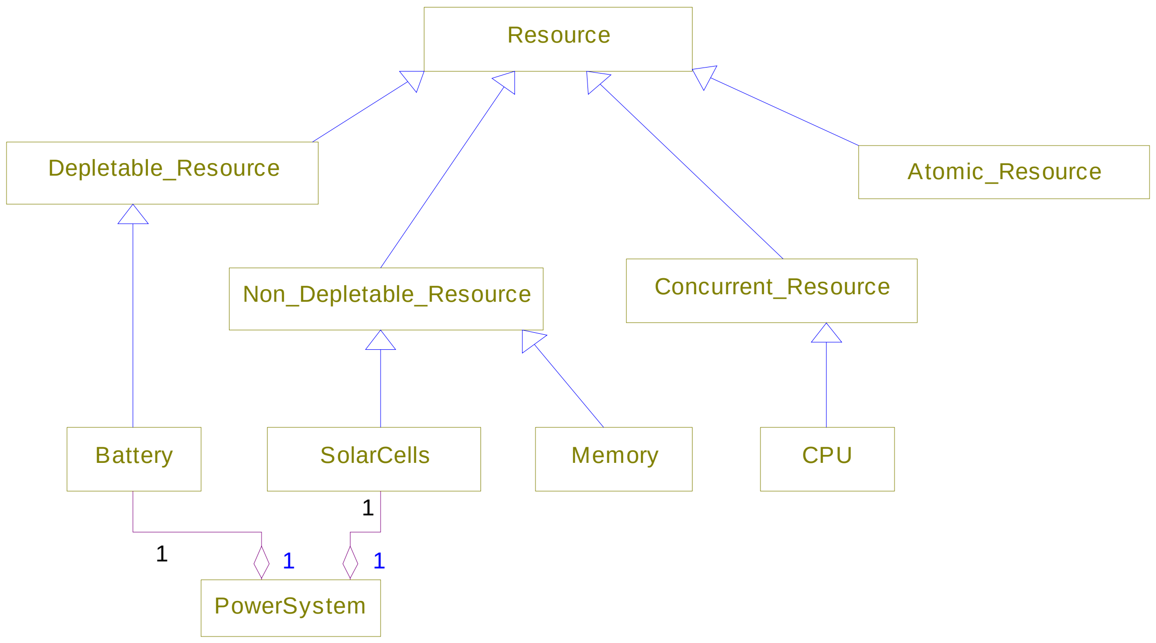


Figure 4.14: The different types of resources.

StereoVision component's usage of memory and time is more significant that its usage of power. The latter can be ignored if it is significantly smaller than the power consumption of other sub-systems.

4.7.3 Different Levels of Resource Prediction

The Functional Layer can provide predictions of resource usage with varying degrees of resolution. Internally, it implements this by having each component provide an estimate, if available, or recursively query its subordinates to obtain a more detailed estimate. The degree of recursion is controlled by the client that requests the prediction. The trade-off between a detailed estimate and a worst-case maximum prediction is that of speed of response vs. computational cost. When the Decision Layer queries the Functional Layer, it can specify the required fidelity of the estimate.

Consider for example the Manipulator class. To properly estimate the amount of energy needed to move the manipulator from point A to point B, it is important to know the current kinematic configuration of the manipulator and the trajectory profiles of each of actuator. The load that each actuator sees is proportional to the amount of power it will consume. The load on each joint varies as the manipulator follows the trajectory. The objects that are queried are not the generic ones, but the specialized concrete classes. In the case of the Manipulator class, it is the object of the specialized class, such as R7Arm, that is queried for the amount of power required for the move. This R7Arm object will use its inverse kinematics to compute the trajectory of each of the joints. Then, each ControlledJoint object is queried for the amount of power they require to complete their respective trajectories under nominal loading. The R7Arm object will then factor arm configuration information with the results returned from the ControlledJoint object. This is an example of a detailed estimate with recursion. Alternatively, the R7Arm can return a worst-case estimate based purely on the Euclidean distance between point A and point B.

4.7.4 Local vs. Global Conflict Resolution

The Functional Layer only manages local atomic and concurrent resources that require fast context switching. Atomic resources are represented within the components of the Functional Layer using binary semaphores. If the component were to occupy an atomic resource for a prolonged period, it is the duty of its superior component or the Decision Layer to coordinate the execution of the subordinate. Otherwise, the operating system scheduler manages the fast switching resource, hiding the shared resource from the rest of the system.

Consider the example of the `Mobile_Manipulator` class presented in Section 4.3.1. When the arm is commanded to move to a point beyond its current reach, the mobile arm will drive the locomotor toward the target to allow the arm to reach it. The implementation of the `Mobile_Manipulator` class does not impose any limits on how much the arm can drive the mobile platform. Now suppose, there was an obstacle in front of the robot. When the arm starts driving the `Locomotor` object, it does not know anything about the obstacle since its attached `Locomotor` object does not have visual sensing. This type of conflict is resolved at the Decision Layer which will command the robot to visually inspect the scene before the command is issued to the mobile manipulator.

4.8 System Control Package

This domain is home for several growing areas of research. Within this package are several sub-packages including ones for neural network control, fuzzy logic control, and behavior-based control. We have laid out some designs for a few classes in the behavior-based control domain.

In behavior-based control, a robot is controlled using behaviors rather than goals or commands. Such control can be implemented using the different granularity levels of the Functional Layer. Depending on the level of interest, one can implement this control at the motion control level, at the sub-system level (such as the mobility or manipulation level), or at the rover level (for multi-rover coordination). Behavior-based control uses a set of robot behaviors that run in parallel but are coordinated using a coordinating function. There are two types of coordination functions to resolve conflicting behaviors: (1) cooperative coordination and (2) competitive coordination.

Figure 4.15 shows the `Behavior` class which provides the core data-structure for behavior types. It attaches to any member function of GPCs or GFCs. A behavior can be started and stopped at any time. A behavior can also encompass other behaviors. This is accomplished through the `Composite_Behavior` class. It is derived from the `Behavior` class but it also aggregates that class. By making the `Composite_Behavior` a child of `Behavior` , CompositeBehavior objects will be of the same type as `Behavior` objects and can hence be used interchangeably. This pattern which we have seen earlier for the `ColorImage` class is known as the composite pattern [40] where the composite object includes several objects but also has the same type as the objects it includes.

The `Composite_Behavior` class has a `Behavior_Coordination` object to resolve behavior conflicts. `Behavior_Coordination` objects resolve conflicts between any behavior types. These constructs can be attached to any operation within the Functional Layer.

There are some COTS packages that implement domain specific infrastructure. Some use an object-oriented framework such as Mobility software from Irobot [2] while others use a procedural framework, such as Saphira [53]. The Saphira package provides an extensive framework for fuzzy-based control of mobile systems. It is desirable to integrate functionality of different domain packages into a single framework to simplify comparative studies of different control and operation approaches.

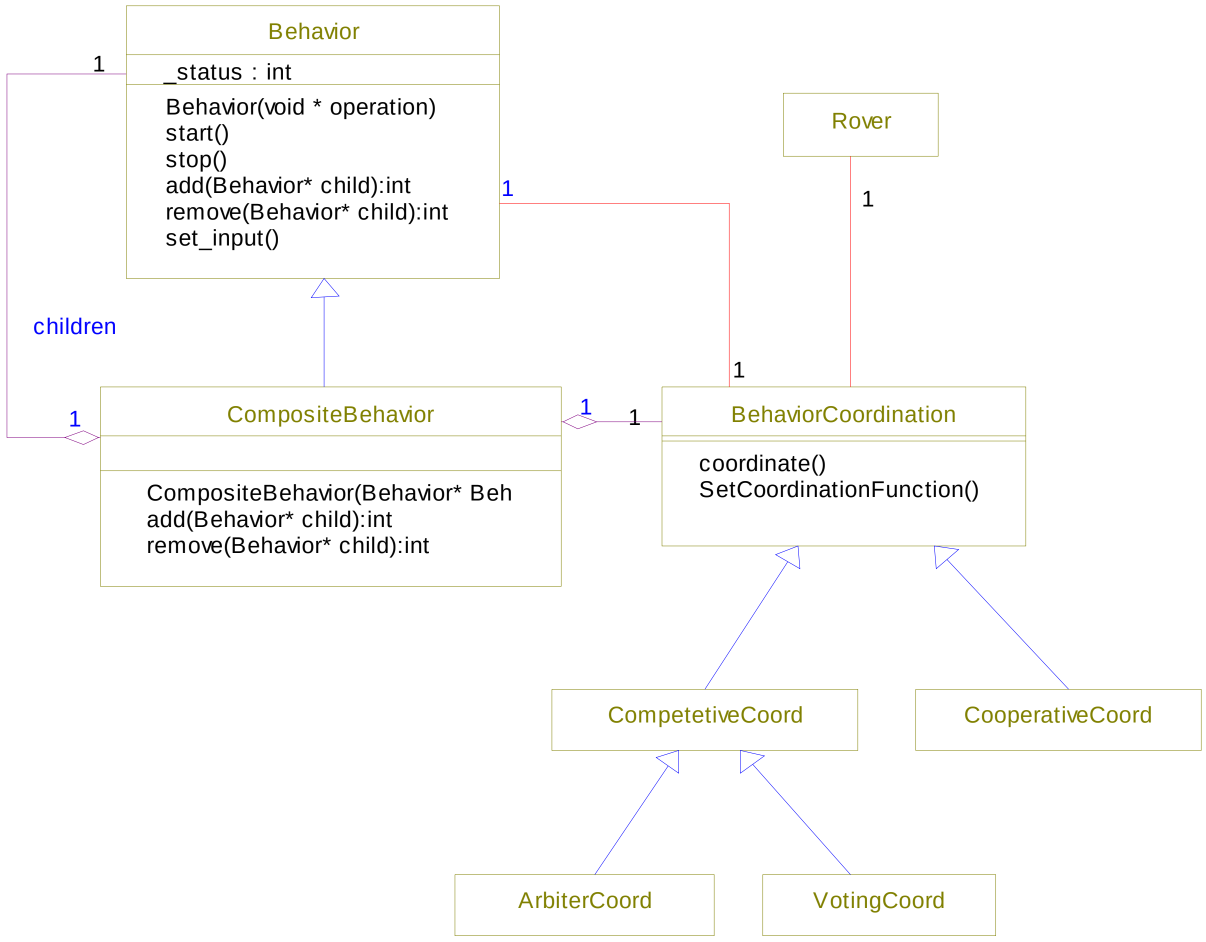


Figure 4.15: Some `Behavior` classes and their relationships.

4.9 Testing, Verification, and Simulation

Each component in the Functional Layer has a test class that derives from it. The purpose of the test class is to test the correctness of the internal implementation as well as the external operation. A verification class carries a series of test and reports a summary of the results.

Another category of classes are simulation classes. These are specialized classes that provide simulation of a component's functionality. For example, a `ControlledMotor` class can be attached to a `SimControlledMotor` class instead of an `R7_ControlledMotor`. The simulation class is derived from its generic counterpart. It provides a software simulation of the functionality of that component. A `move()` operation to the simulation class will, for example, spawn a task that will periodically update the current position mimicking the velocity profile supplied to the move operation. Queries about the current motor position will return an ideal position under no loads. The simulator class can implement simulations to varying levels of fidelity. They can also be adaptor classes that attach to a high-end simulator such as DARTS/Dshell [17].

Another example is the `Sim_Camera` class. In this class, the `acquire_image()` operation can be overridden by reading a static image from disk, or even obtaining a snapshot from a terrain simulator. Each operation of that class will have to provide some sort of simulated functionality. To the rest of the application, whether a `Camera` is attached to a SimCamera object or a specialized object, such as `R7Camera`, should make no difference. Simulation classes are important during development since many parts of the system will be in debug or development mode when others are being developed.

Part III

The Decision Layer

Chapter 5

Decision Layer Background

This Chapter contains an introduction to relevant Decision Layer background material. It defines terminology that is used in this document to describe the Decision Layer portion of the CLARAty architecture and provides background information on past approaches to planning and executive systems for robotic platforms.

This chapter is organized in the following manner. First, Section 5.1 provides background definitions. Next, Section 5.2 describes and compares declarative and procedural knowledge representations. Finally, Section 5.3 gives a brief overview of different planning and scheduling techniques, Section 5.4 gives a brief overview of different executive techniques, and Section 5.5 describes several examples of planning/scheduling and executive systems.

5.1 Decision Layer Definitions

The input to the Decision Layer is usually a set of goals to be achieved. A *goal* is defined as a constraint on a particular state-variable over a certain time interval. A *state-variable* is used to represent and reason about actual states of the robot and its subsystems (e.g., memory usage, mast position, robot orientation). A *constraint* on a state-variable dictates what particular state the state-variable must be in over a certain time interval. For instance, suppose a rover has a spectrometer as one of its science instruments and the spectrometer is typically turned off when not in use. A state variable representing if the spectrometer is powered-on is shown in Figure 5.1. In this figure, we are assuming that no plan activities or goals have been scheduled that affect this state variable so it reflects the default off-state for the spectrometer for the plan horizon.

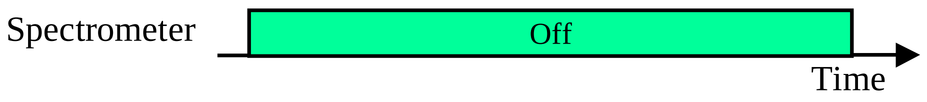


Figure 5.1: State variable showing on/off state of spectrometer instrument with no spectrometer activities scheduled.

If a science goal is scheduled to take a spectrometer reading, then this state will need to be modified so that the spectrometer is turned on at the appropriate time.[1] A sample goal for a spectrometer read scheduled on this timeline is shown in Figure 5.2, where the spec-read goal is requiring the spectrometer to be on from time T1 to time T2.

[1]This requirement would be part of the goal definition for a spectrometer read.

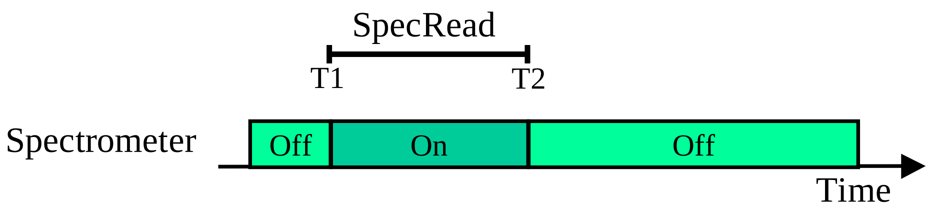


Figure 5.2: Goal for performing a spectrometer read that requires the spectrometer to be on over a certain time period.

State variables can also represent aggregate resources such as power or memory. Figure 5.3 shows how a spectrometer read might affect a memory state-variable by increasing the amount of data stored in memory.

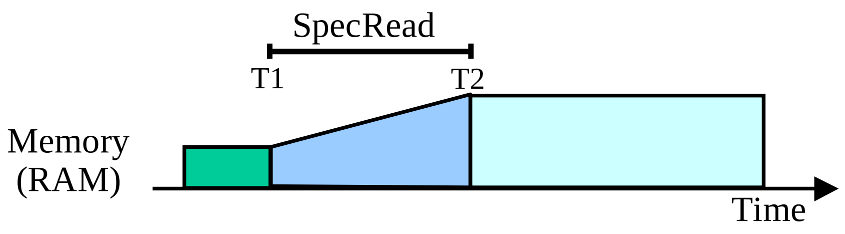


Figure 5.3: State variable showing resource level of onboard memory.

Many planning and scheduling systems use the term *activity* to represent a high- or low-level action that takes place over a certain time interval. This action can represent a goal, exert constraints on state variables, represent an abstract activity (that must be further broken down), and/or correspond directly to a low-level command that should be passed directly to the Functional Layer. Thus an activity is a more general structure that subsumes the goal structure explained above.

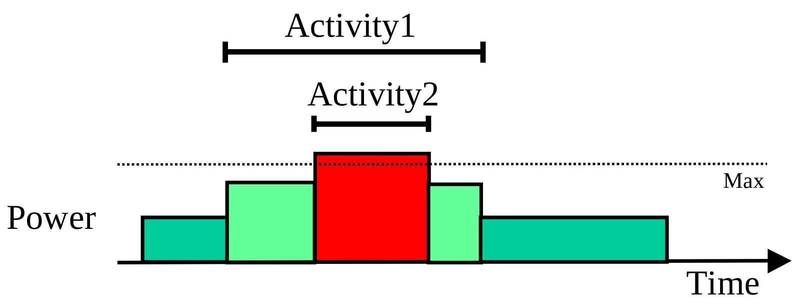


Figure 5.4: Two plan activities and their effects on a power resource.

Figure 5.4 shows two different activities and their resulting effects on a resource timeline representing power. The red (or dark middle) part of the power timeline shows the portion of time where power is being oversubscribed (i.e., the plan is requiring more than the max amount of available power).

We call the structure used to represent the plan in the Decision Layer a *goal-net*, which corresponds to the temporal constraint network of goals, activities, and tasks. A simple goal-net is pictured in Figure 5.5, which shows several activities and the set of temporal constraints on the activity start and end times.[2] Temporal constraints usually reflect that the start/end time of one activity must come before the start/end time of another activity. They can also be more detailed in that a constraint may have a time bound on how far apart the activities can be (e.g., the end-time of activity A must end between 0 and 5 seconds before the start-time of Activity B). Low-level activities in the goal-net usually correspond directly to commands that are dispatched to the Functional Layer during plan execution.

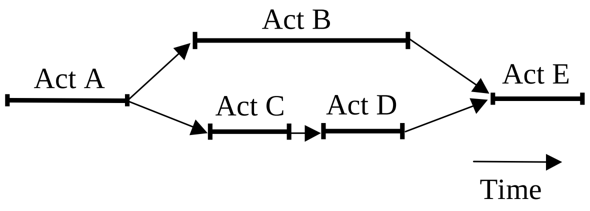


Figure 5.5: Sample goal-net for several activities.

We use the term *elaboration* to represent the decision process of achieving a goal, i.e., creating an appropriate goal-net [51]. This process can be performed by a planner/scheduler, executive, scripting language, straight code, or a combination of such software.

5.2 Declarative vs. Procedural Knowledge

The Decision Layer handles two main types of knowledge, declarative and procedural. One or a combination of these two representations is used to build a model of the domain and its relevant resource and operation constraints. This model will contain such information as how much power the robot has, what activities must be performed in a particular sequence, the average and/or maximum duration of activities, etc.

Many domain models are built using *declarative* knowledge. When employing a declarative representation, knowledge about the domain is given but the exact method of using the knowledge is not specified. In other words, declarative knowledge can be seen as data to a program but does not contain any of the program logic. Most planning and scheduling systems primarily use declarative knowledge to build models. An example of a declarative representation of two planning activities for loading a vehicle and driving a vehicle are shown in Figure 5.6. Using a typical planning representation, these activities are defined by specifying their preconditions (which must be true before the activity is executed) and their effects (which are true after the activity is executed). Note, that the order in which the activities must be used or a description of a specific situation in which the activities must be used is not directly specified.

Domain models can also be built using *procedural* knowledge. When utilizing a procedural representation both knowledge and the necessary control information to use that knowledge is specified. In other words, the knowledge can be viewed as a program where instructions on how to use the knowledge are encoded within the knowledge itself. Most execution systems utilize a procedural representation where constructs such as if/then conditions and loops can be easily specified. In Figure 5.7, we show a procedural rule

[2]The goal-net pictured in Figure 5.5 doesn't show connections between high- and low-level activities. Some activities in a goal-net may be abstract (or high-level), while other activities are low-level and unable to be broken down further. For instance, in Figure 5.5, Activity E might be further broken down into several sub-activities. Temporal constraints can be drawn between any two types of activities regardless of their abstraction level. Constraints on a parent activity automatically apply to any of its sub-activities.

Load_vehicle(Obj,Veh,Loc)

Preconditions: at-obj(Obj,Loc1), can-carry(Veh,Obj),
at-vehicle(Veh,Loc)
Effects: inside-vehicle(Obj,Veh), ¬at-obj(Obj,Loc)

Drive_vehicle(Veh,Loc1,Loc2)

Preconditions: at-vehicle(Veh,Loc1), near(Loc1,Loc2),
enough_fuel(Veh,Loc1,Loc2),
enough_time(Veh,Loc1,Loc2)
Effects: at-vehicle(Veh,Loc1), ¬at-vehicle(Veh,Loc2)

Figure 5.6: Declarative activity definitions for loading and driving a vehicle.

If: object needs to be moved,
vehicle is near object,
vehicle can carry object,
new location is in driving distance,
vehicle has enough fuel,
there is enough time left in day

Then: load the object into the vehicle

Figure 5.7: Procedural rule definition for when to load an object into a vehicle.

definition for our previous load-vehicle activity. In this representation, the exact logic on when to apply the activity is given.

5.3 Planning and Scheduling Overview

This section gives a brief overview of AI planning and scheduling algorithms, especially those tested on robotic domains. The role of planning and scheduling in the CLARAty architecture is to generate a temporally constrained plan of actions based on an input set of goals and a domain model.

We can divide algorithms of this general area between those that perform planning and those that perform scheduling. *Planning algorithms* are typically designed to determine a course of action (i.e., plan) from a set of goals. These systems are usually aware of what activities can be placed in the plan and what are the preconditions and effects of the activities. *Scheduling algorithms* assign times and resources to plan activities and can handle such constraints such as resource limitations, precedent relationships, due dates, etc. In addition, planning and scheduling systems often attempt to optimize one or more objective functions that directly measures the quality of a plan. These functions may use factors such as cost, time, and resources used to evaluate plans.

Planning is usually done through search or with hierarchies. One common search technique employed by AI planning systems is using precondition and effect analysis with a forward or backward-chaining search. When searching backwards (i.e., from goal-state to initial-state) this search process is called subgoaling. During this search, goals can be examined in a linear or non-linear order. If a linear order is imposed, then all of the subgoals of the current goal must be achieved before examining the next goal. Another common

```
Activity mast_placement {
    position x, y, z;
    angle heading;
    duration = 300s;
    reservations = mast, mast_sv change_to “deployed”,
      health_sv must_be “exec”,
      drive_motors, day_night_sv must_be “day”;
}
```

Figure 5.8: A planning activity for rover mast placement.

planning technique is called hierarchical-task network (HTN) planning [29]. HTN planners decompose high-level goals into low-level activities. For both these techniques, plans can be represented in a totally-ordered or partially-ordered fashion. A totally-ordered plan requires all plan steps to be ordered with respect to each other. A partially-ordered plan contains only necessary orderings and allows some plan activities to be executed in parallel.

Scheduling systems must assign resource usage and time values to the set of planned activities. These assignments must obey all relevant rules or constraints that specify domain information such as temporal relationships, resource limitations, valid state transitions, etc. Scheduling techniques fall into two main categories: constructive methods and repair methods. Constructive methods incrementally extend partial schedules until they are complete [38, 62, 75]. Repair methods use techniques such as*iterative repair* to fix conflicts in a complete but inconsistent schedule [58, 88, 24]. Search methods can also use a fixed timepoint representation, where all activity timepoints are grounded to exact values (i.e., total time commitment), or a flexible timepoint representation where activity timepoints are associated to legal time windows in which they can occur.

Planning and scheduling systems usually employ a declarative domain representation for domain activities, resources and states and any relevant constraints over those entities. As an example, an activity definition for placing a rover mast is shown in the ASPEN planning system [39] format in Figure 5.8. This particular activity definition contains several parameters, an expected activity duration, and a number of resource and state constraints. For instance, this activity requires that the mast state-variable be in the state of “deployed.” A domain model includes information such as activity definitions and requirements, resource and state definitions and constraints, temporal constraints, predicted resource usages, predicates states of the world, etc.

5.4 Executive Overview

The job of an executive system is to take the final plan from the planning/scheduling system and generate the necessary actions. The executive provides event-driven behavior by expanding an abstract plan into low-level commands, executing and monitoring these commands, and handling any exceptions or unexpected behavior. Unlike a planner, the executive has no future projection capabilities. Instead it is intended to be reactive to the environment and the robot’s current state.

There are many challenges at this layer including handling the execution of concurrent activities and handling exceptions that may cause the current plan to be modified and/or may cause a non-local flow of control. The executive must be able to respond conditionally to the environment and be able to modify the plan accordingly in a timely fashion. Actions executed by the executive can include conditional, iterative and even recursive activities.

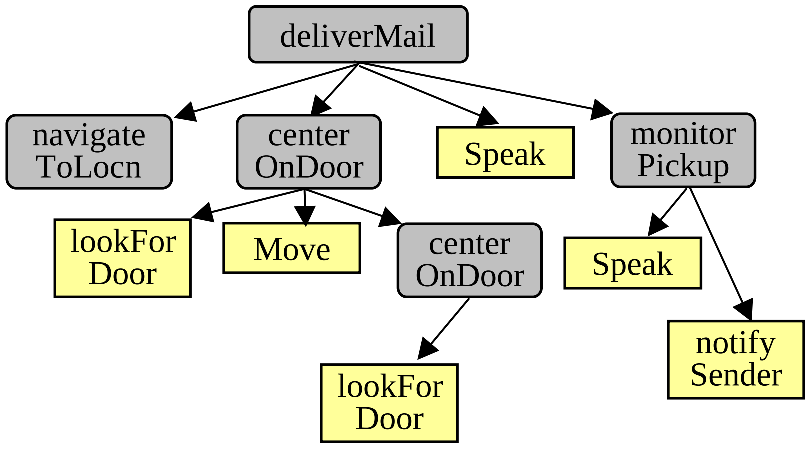


Figure 5.9: A task tree for delivering mail.

Many different representations have been used for Executive systems. The term *task* is commonly employed by Executive systems and represents a concept similar to an activity. Tasks generally correspond to low-level activities that are decomposed and scheduled by an executive. The term *task tree* is used to describe the tree structure that is produced when tasks are broken down into lower-level tasks. A task tree for an executive can be considered similar to a plan of activities for a planner or scheduler.

The Task Description Language (TDL) [73] uses task trees to represent information for task-level control. Figure 5.9 shows an example task tree for a robot delivering mail. The actual task definition is shown in Figure 5.10. A task tree encodes parent/child relationships and synchronization constraints between tree nodes, and associate exception handlers. The Execution Support Language (ESL) [44] uses a set of macros implemented in Common LISP to represent task information. It provides facilities for contingency handling, resource management, and task management. Like TDL and ESL, most executive systems have some capability for representing procedural type information in the form of specific rules or macros.

```
Goal deliverMail (int room) {
    double x, y;
    getRoomCoordinates(room, &x, &y);
    spawn navigate to Locn(x,y);
    spawn centerOnDoor(x,y)
        with sequential execution previous,
            terminate in 0:0:03.0;
    spawn speak (“Xavier here with your mail”)
        with sequential execution centerOnDoor,
            terminate at monitorPickup completed;
    spawn monitorPickup()
        with sequential execution centerOnDoor;
}
```

Figure 5.10: TDL definition for *deliverMail* task.

Depending on the overall system, executives can have more or less functionality. At a minimum, an executive acts as an activity dispatcher to low-level controllers, however no further procedural expansion of activities is performed. In this mode, the executive simply ensures that activities are executed at the

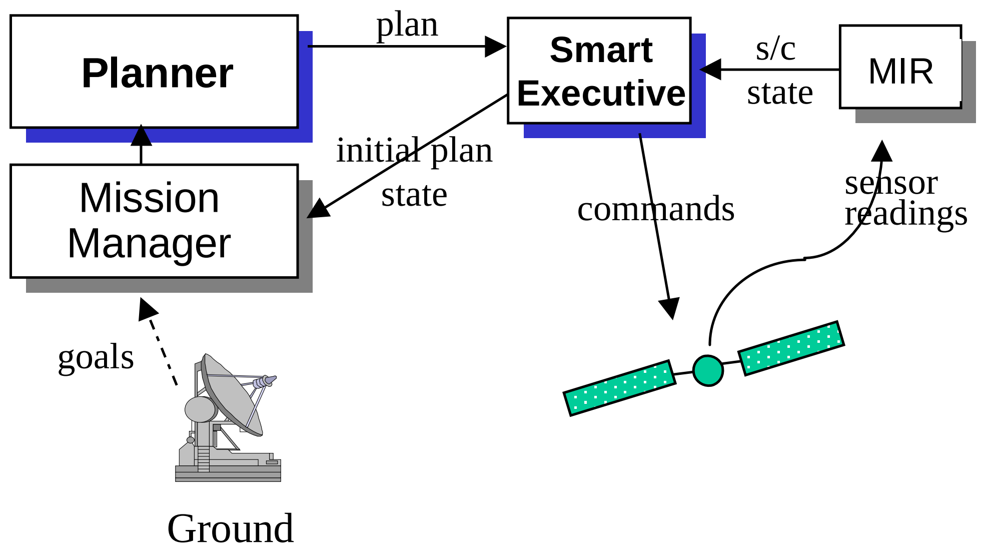


Figure 5.11: The Remote Agent Architecture.

appropriate times and/or when the appropriate conditions are true. At a maximum, the executive acts as a procedural engine for sequence generation. In this mode the planner is bypassed and the executive directly breaks high-level goals into commands, executes those commands and modifies the command sequence based on state feedback.

5.5 Planner and Executive Examples

In this section, we will briefly describe several examples of planning/scheduling and executive systems that have been successfully used for robotic applications.

5.5.1 RAX: Remote Agent Experiment

The Remote Agent Experiment [16] was flown on the NASA Deep Space One (DS1) mission. Its primary purpose was to provide an onboard demonstration of spacecraft autonomy. In particular, this experiment demonstrated the ability of an AI system to respond to high-level spacecraft goals by generating and executing plans onboard the spacecraft.

Two of the systems which comprise RAX can be easily mapped to the previously discussed planning and executive technologies. These systems are the onboard planner-scheduler, RAX PS, and the onboard multi-threaded Smart Executive. The RAX architecture is shown in Figure 5.11. RAX is comprised of four components, which include a Mission Manager for handling high-level goals, a Planner/Scheduler for creating plans, a Smart Executive for executing plans, and a Mode Identification and Repair (MIR) component, which determines the state of the overall spacecraft and can suggest repairs when exceptions occur. For this discussion we will only discuss the planner and executive modules.

The RAX PS planner takes as input a schedule request and produces a flexible, temporal schedule for execution by the Smart Executive. The schedule is constructively produced using a heuristic backtracking search that must account for resource and temporal constraints as well as any complex flight safety rules on activity interactions.

The executive component of RAX was supported by a version of the Execution Support Language (ESL) [44]. ESL executes a plan by decomposing the high-level plan activities into primitives, sending out commands, and monitoring progress based on direct feedback from the spacecraft subsystems. If some task cannot be achieved in the RAX scenario, ESL may try an alternative method that fits within the plans temporal flexibility. If ESL is unable to execute or repair the current plan, it aborts it and requests a new plan from the RAX PS planner.

In this architecture the planner and executive have different representations and strictly operate on different layers of abstraction. Planning is performed in a batch fashion where the planning system only runs when required. If re-planning is required, the spacecraft must be safed until a new plan has been generated, which can often be a significant amount of time.[3]

5.5.2 CASPER

The CASPER (Continuous Activity Scheduling Planning Execution and Replanning) system utilizes a*continuous planning* approach to achieve high-level goals while still respecting resource and temporal constraints [24]. Rather than considering planning a batch process in which a planner is presented with goals and an initial state, the planner has a current goal set, a plan, a current state, and state projections into the future for that plan. At any time an incremental update to the goals or current state may update the plan. This update may be an unexpected event or simply time progressing forward. The planner is then responsible for maintaining a plan consistent with the most current information. The current plan is the planner's estimation of what it expects to happen in the world if things go as expected. However, since things rarely go exactly as expected, the planner stands ready to continually modify the plan. Iterative repair techniques enable incremental changes to the goals, initial state or plan, and then iteratively resolve any conflicts that may arise.

CASPER encompasses the job of the planner and handles the job of a minimal executive where activities are dispatched and state updates are handled. The CASPER planner operates on a shorter time scale than most other planning systems. In CASPER, planning is intended to be performed quickly and incrementally. State changes are monitored and the plan is never allowed to get out of sync for very long. Currently there is no functionality in CASPER for more sophisticated executive capabilities such as contingency handling and error recovery, however as discussed in Section 6.5, these capabilities can be provided by integrating CASPER with an executive system.

CASPER has been applied to a wide range of NASA application domains. Figure 5.12 shows CASPER being used as part of the Rocky 7 architecture to perform planning and re-planning based on input science goals [82]. Other domain examples including performing landed operations scenarios for the ST4 mission [24], performing science operations for a team of rovers [31, 32], and controlling Deep Space Network (DSN) antenna operations [37]. CASPER has also been baselined as part of the flight software for the 3-Corner Sat (3CS) mission [78] to be operated by the Colorado Space Grant Consortium.

5.5.3 TDL

The Task Description Language (TDL) is an executive development framework that was built as an extension to the C++ programming language. TDL is intended to simplify the development of robot control programs by providing support for task control functionality [73]. Capabilities supported by TDL include task decomposition, task synchronization, execution monitoring, and exception handling. This language (in combination with a Task Control Management/TCM library) enables a user to easily manage task-control aspects of a robotic system. An executive developed in this environment can expand abstract goals into low-level commands interpretable by low-level controllers, execute these commands, monitor their execution, and handle arising exceptions.

[3]In the RAX experiment, plan generation was typically allocated four hours.

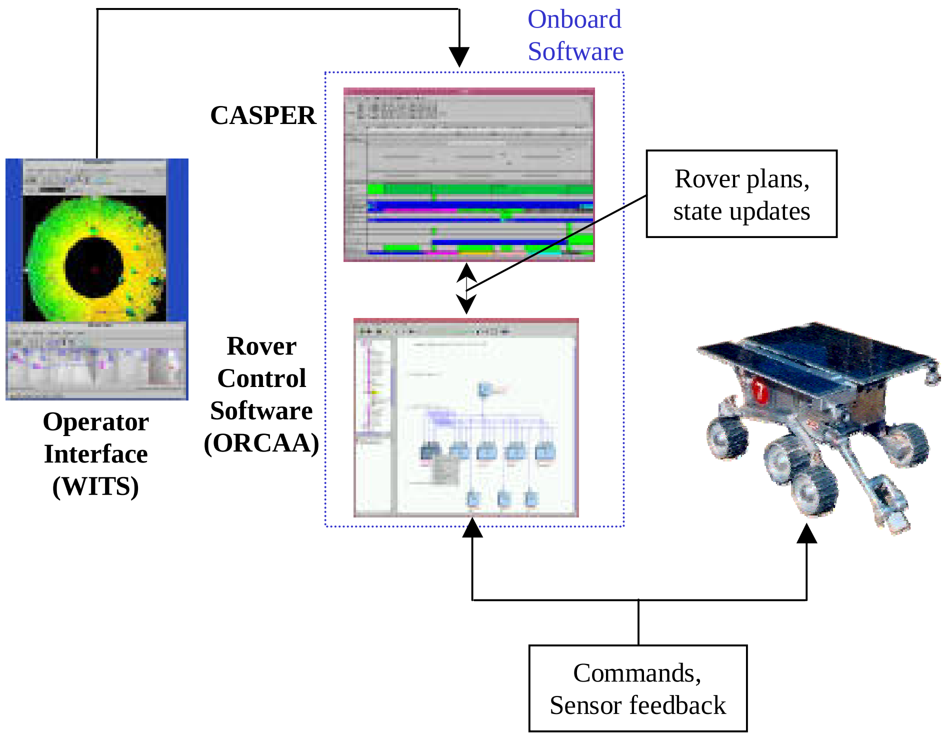

Onboard Software
CASPER
Rover plans, state updates
Rover Control Software (ORCAA)
Operator Interface (WITS)
Commands, Sensor feedback

Figure 5.12: Using CASPER continuous planning system to generate command sequences for Rocky 7 rover.

As mentioned previously, TDL uses task trees to represent information for task-level control. A task tree encodes parent/child relationships as well as synchronization constraints between tree nodes and exception handlers associated with certain tasks. TDL-based control programs operate by creating and executing task trees. Task trees are generated dynamically so the system can use current perception to make decisions about what nodes should be added to the tree and how to correctly parameterize their associated actions. Actions can include conditional, iterative, and recursive code. As compared to other executive approaches, TDL includes a wider range of task synchronization constraints that enable intricate control strategies to be encoded. TDL has been demonstrated on a number of different robotic platforms including the Nomad robot used for the Antarctica 2000 initiative [60], the Bullwinkle RWI robot used for Mars Autonomy navigation [74], and the Xavier robot, which has been used for numerous autonomous mobile robot projects at CMU [72].

5.5.4 Other Related Work

These are a number of examples of system implemented for robot control that use both planning/scheduling and executive technology. For example, the Atlantis [43] and 3T [18] approaches both consist of three main components: a deliberative planner, a reactive feedback mechanism, and an executive (or sequencing component) that connects the other two components and handles plan execution. Both of these architectures integrated high-level planning and reactive behavior asynchronously and were tested on mobile robot domains. The LAAS-CNRS lab have also developed a robot control architecture that is intended to support autonomous capabilities [4]. This architecture is also composed of three levels: a decision level, an execution level, and a functional level which are integrated to allow both planning and reactive capabilities. The decision level can also be further decomposed into additional layers depending on application needs.

Different instantiations of this architecture have been tested on several indoor and outdoor robots.

Furthermore there are numerous examples of particular planning/scheduling and executive systems that do not comprise a complete architecture but provide different elements of robot control. A variety of different planning and scheduling systems have been created to provide the capabilities for creating a robot plan to satisfy a set of high-level goals based on some world model. We have already mentioned two such planners, CASPER [24] and the DS1 RAX-PS planner [50]. Other examples of planning systems for robot applications include CPS, AP and the IxTeT planner. The Contingent Planner/Scheduler (CPS) approach was developed to schedule scientific operations for rovers using the Contingent Rover Language (or CRL) [21], which allows both temporal flexibility and contingency branches in rover command sequences. An early version of this scheduler was used in field tests for the Marsokhod rover. The Adversarial Planner (AP) [18] is part of the 3T architecture mentioned previously. AP is directed towards multi-agent control where robots are not fully autonomous (i.e., humans are in the loop) and can also reason about uncontrolled agents that might affect certain world states. The IxTeT planner [54] is utilized in the LAAS-CNRS robot architecture and searches through a tree of partial plans until no plan flaws remain. IxTeT can also be used to successfully merge several plans into one consistent plan.

Similarly there are numerous examples of executive or sequencing system that have been utilized for executing a robot plan or sequence of commands. Many of these systems have been demonstrated in conjunction with a planning system, however that is usually not a requirement. We have already mentioned both the TDL executive [73] and the ESL executive [44]. Additional executive examples include RAPS, PRS, and PRS-lite. Reactive Action Packages (RAPS) [34] defines methods for decomposing tasks into subtasks, for detecting successful achievement of tasks, and for responding to changes. RAPS is particularly geared towards defining reactive behaviors and real-time response. The Procedural Reasoning System (PRS) [46] is based around the concept of a procedural reasoning expert and represents explicitly the psychological attitudes of belief, desire and intention for a mobile robot. Plans or intentions formed by the robot need only be partly elaborated before action is taken, and the robot is continuously reactive to the environment. PRS-lite [64] is a reactive controller that is loosely based on PRS, however it has a richer goal semantics and a generalized control regime, and it was specifically designed to control behaviors implemented using fuzzy logic.

The intent of our architecture is that any of the above approaches can be easily integrated into one framework. CLARAty supports a variety of planning and executive approaches and is designed to be flexible enough so that more tightly integrated planning and execution approaches can be utilized as they become available. Current research into such systems will be further discussed in the next section.

Chapter 6

Decision Layer Description

This chapter contains a description of the proposed CLARAty Decision Layer and a discussion of related issues. It is organized in the following manner. First, Section 6.1 gives an overview of the Decision Layer. Next, Section 6.2 lists the objectives the Decision Layer must meet. Section 6.3 describes how the state of the Decision Layer is maintained. Next, Sections 6.4 and 6.5 describe motivation and work on integrating planning and executive systems. Section 6.6 describes the proposed Decision Layer framework, and Section 6.7 describes how the Decision Layer interfaces to the Functional Layer. Finally, Section 6.8 describes how the Decision Layer is related to the JPL Mission Data System architecture effort.

6.1 Decision Layer Overview

The Decision Layer consists of a hierarchical structure that overlays the Functional Layer. An abstract view of the Decision Layer was shown in Figure 1.5. The primary responsibility of this layer is to break high-level goals down into a detailed plan of activities that successfully coordinate Functional Layer capabilities in achieving the goals. This plan must also obey any relevant domain or mission constraints, such as resource limitations or instrument operation rules. As shown in Figure 1.5, the Decision Layer can be thought of as triangle, which represents the "robot planning space." Here, a few high-level goals are elaborated into a detailed network of goals and activities that represent the current plan.[1] The elaboration process terminates at subgoals that are not designed to be further decomposed. During elaboration, predicted resource usage for a goal is obtained by querying the appropriate objects in the Functional Layer. The network of activities shown outside of the Decision Layer triangle represents the region of decision-making that is outside the bounds of that particular robot. These goals may map onto other robots, or be within the domain of the overall mission strategy.

The top portion of the red triangle (shown in darker red) is the region of the robot planning space which is handled primarily through planning functions. The bottom portion of the red triangle (shown in lighter red) is the region of the robot planning space which is handled primarily through executive functions. The line between these two regions is fuzzy since executive and planning processes may be tightly coupled and may even shared the same representation and plan database, as discussed in Section 6.4.

The bottom fringe of this activity network is where the Decision Layer interfaces with the Functional Layer. This interface point is shown by the dashed back line (called "The Line"). During plan execution, capabilities in the Functional Layer must be called to achieve each of these activities, and results of these actions are monitored to allow the plan to be modified due to changing events or conditions. This interface

[1]This network is called a Goal Net, and was explained in Section 5.1.

line is flexible and may be moved up or down depending on how much control and elaboration the Decision Layer is responsible for. This flexibility is further explained in Section 6.7.1.

6.2 Decision Layer Objectives

The CLARAty Decision Layer has a number of important objectives:

- Provide a capability for creating a detailed, flexible plan of activities for achieving high-level goals.
- Provide a capability for monitoring plan execution and dynamically modifying the plan if necessary.
- Provide a framework for using different types of planning and executive functionality.
- Enable planning and executive functions to be used at different levels of abstraction.
- Provide the flexibility needed for different research and flight projects.
- Allow for easy development of Decision Layer capabilities for different applications and robotic platforms.

The main objective of the Decision Layer is to generate plan sequences for achieving high-level goals and when possible, to do so in an optimal fashion. The Decision Layer can be viewed as a high-level plan generation engine that drives the lower-level control algorithms contained in the Functional Layer. Planning and scheduling mechanisms are provided in this layer that can construct action sequences that will successfully achieve a set of high-level goals without violating any robot resource or operation constraints. The Decision Layer must also be flexible enough during execution to respond to a dynamic environment, and make plan changes accordingly, to ensure plan success even when conditions change. Furthermore, the Decision Layer must enable domain information such as resource and state constraints to be easily represented and utilized by the final system.

Another objective of the architecture is to provide a framework for using different types of planners and executives. We do not intend to tie our architecture to one particular style of planning or execution and would like different types of planning and executive systems to be easily utilized in the CLARAty framework. The Decision Layer can be seen as loosely corresponding to the top two levels of a traditional three level architecture [45]. These levels hold software that performs deliberative planning and software for reactive plan-execution. An overall system using this approach operates by generating a plan, which is then passed off to an execution agent that further details the plan and monitors its execution. This type of process can be defined as a batch mode of operations where planning is typically considered an off-line process that is only called periodically when there is enough time and resources available. If a plan becomes invalid during execution, it is up to the executive to either fix the current plan by using a limited set of fixes or to simply fail gracefully and halt the system until the planner can create a new plan.

For many situations, the approach taken by the traditional three level architecture is insufficient. Recent research has moved towards creating systems where planning and execution are more closely tied together [8, 26, 63, 24]. Typically planners are used in the batch-mode described above and operate on long-term projections. Executives are used for more reactive, near-term behavior. However, it is sometimes beneficial to use planning functions to refine the near-term plan and/or to use executive functions to elaborate part of the future plan. Enabling these capabilities will allow for increased functionality and increased robustness at the high-level of the architecture. Thus, we want to provide flexibility in the Decision Layer to use planning and executive functions combined at different levels of abstraction and temporal granularity, as well using them separately when warranted by the domain or user.

In addition to planner and executive flexibility, we want other forms of goal elaboration (i.e., scripting, code) to be supported. System designers may want certain goal types to be handled by a specialized piece of software as opposed to a more general-purpose planner or executive. For some goals, it may be more appropriate to use macro definitions or specific pieces of code to break down the goals into activity plans.

This architecture is designed to fulfill needs of robot research projects, in and out of NASA, as well as providing a robot autonomy framework for future NASA flight projects. Thus, we want to ensure CLARAty provides the flexibility needed for these different projects. In particular, we want to allow different levels of capabilities to be easily implemented and/or accessed depending on the user and the intended application. Furthermore, we want the develpment of Decision Layer capabilities to be straightforward so that CLARAty can handle goals for different robotic platforms and/or high-level mission strategies. Users should be able to easily create domain models, which describe the robotic platform and its relevant resource and state constraints, as well as, easily modify these models as their application domains evolve.

These objectives will be accomplished through the following approach. First, we intend to implement a Decision Layer framework that allows both separate and tightly-coupled planning and executive components. Furthermore, this framework will allow domain knowledge to be represented as either declarative or procedural information. Second, we intend to leverage off of existing planning/scheduling systems and executive systems that have been successfully utilized for robotic applications. Third, we intend to leverage off of software being created as part of the JPL Mission Data System (MDS) Project, which was briefly discussed in Section 6.8, and has been building a general software architecture to support future NASA missions. Last, we will validate the CLARAty Decision Layer by implementing the architecture on different robotic platforms - both real and simulated.

6.3 Decision Layer State

The state of the Decision Layer is maintained in an*Activity Database.* This database holds all activities and temporal constraints that have been added to the goal-net as well as all relevant state and resource timelines for the particular application. This database is intended to hold information for both planning and executive functions and can be modified by either of these system types. Resource and state timelines hold both future plan projections and the execution history of the system.

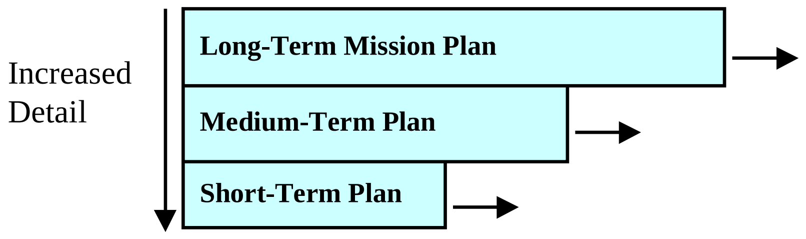


Figure 6.1: Hierarchical plan projections.

The current plan can also be represented within the activity database at different layers of abstractions by using hierarchical plan projections, as shown in Figure 6.1. In such an approach, the long-term planning horizon is planned only at a very abstract level. Shorter and shorter planning horizons are planned in greater detail, until finally at the lowest level, the plan is fully expanded. The idea behind this type of hierarchical approach is that only very abstract projections are made over the long-term and that detailed projections are only made for the short-term plan since long-term prediction is often difficult due to limited computational resources and unknown future information [24].

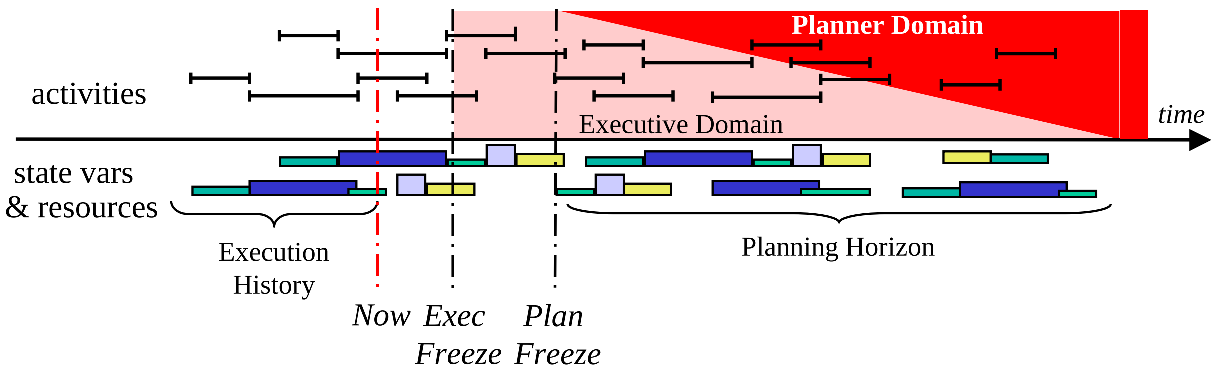


Figure 6.2: Activity domain of planner and executive.

6.4 Merging Planning and Executive Functionality

In the previous chapter we described a number of systems that concentrated on either goal-driven or event-driven behavior. Goal-driven behavior enables a robotic system to accept high-level goals rather than low-level instructions. This capability is often provided though automated planning and scheduling techniques. Event-driven behavior enables a robotic system to quickly react to changes in its environment and modify its command sequence appropriately. This capability is usually provided through an executive system. As discussed previously, these two capabilities are traditionally combined in a three-tiered approach.

Separating these capabilities is actually limiting for many applications since functionality from the planning or executive layers can only be used at a certain level of abstraction and for a particular time scale. In the typical three-tiered framework, the planning system usually works at a very high-level of abstraction and on a long-term time scale. The executive component works on a much shorter time scale and is able to react quickly to unexpected faults, however it has no knowledge of long-term predictions. These two levels also typically use different representations, which further separates their functionalities. Planning systems typically represent domains in a declarative fashion, which enables a planner to perform global reasoning about plan optimality. Executive systems utilize a procedural representation, which enables conditional behavior driven by current state information. Though this separation of capabilities and representation works for some applications, there are many situations where it would be beneficial to have event-driven capabilities available at a higher-level of abstraction. For instance, sometimes a conditional reaction or looping behavior may be required in a high-level activity or in an activity scheduled significantly in the future. The sooner such an activity is properly expanded in a plan, the sooner information on its state and resource usage can be propagated and reasoned about. Furthermore, it would often be beneficial to have goal-driven capabilities available on a short-time scale. For example, there may be low-level activities whose resource usage we want the planner to track and reason about during its search process. If the executive makes a decision about expanding an activity, the planner may be able to optimize over that decision by performing a global analysis on how that expansion fits into the plan. Without planning-type capabilities on a shorter-time scale, many activity resource and state effects must be done using a worst-case approximation which can significantly affect plan optimality.

The desired behavior of a combined planner and executive is shown in Figure 6.2. In this approach, the planner and executive would operate on the same set of activities and timelines and all capabilities would be allowed on both near and far-term activities. The shaded activity areas of the figure show where the executive and planner would be active. The executive would be primarily active on a short-term basis but could be used to refine long-term plan activities. Similarly, the planner would be primarily active on a long-term basis but could be used to plan for short-term activities as well. A separate module would decide what

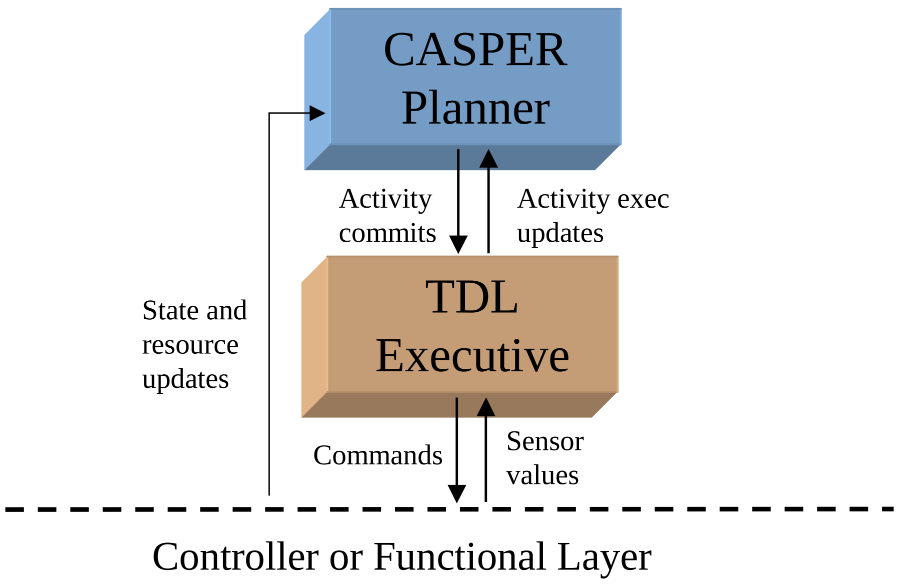


Figure 6.3: Simple integration of CASPER planner and TDL executive. Both systems treat each other as a black box.

functionality was being used on what activities and would synchronize the two sets of capabilities. Such a closely integrated approach to planning and execution will improve robotic system responsiveness as well as increase representation flexibility.

In addition, some executive behavior will exist and/or be duplicated in the CLARAty Functional Layer. In Section 6.7.1 we explain how some pieces of functionality can exist in both layers. This approach allows flexible use of the architecture for different problems and by different users. Some executive-type behaviors may be better suited for residing in the Functional Layer, such as canned sequences or the tight coordination of multiple subsystems. Other executive-type behaviors might be implemented in both the Decision Layer and Functional Layer and instantiated in one layer or the other for different problems. For instance, some users may want localization activities planned out in the Decision Layer while other users may want localization actions only handled by the Functional Layer. It is important to note that Functional Layer capabilities are only locally optimal by design, while Design Layer capabilities allow for coupling and optimization across subsystems that are otherwise de-coupled.

6.5 Current Work on Planner/Executive Integration

Currently, there are several sets of work being performed to provide the first step toward such an integrated system. One set of work is to integrate the CASPER planning system with the TDL executive system [36]. A preliminary integration has already been demonstrated where CASPER and TDL are connected in the typical three-level architecture approach. For this integration, which is shown in Figure 6.3, a set of high-level goals is input to CASPER which then creates an abstract plan. This plan is forwarded to TDL which further breaks down the plan into low-level commands. State updates are sent back to TDL which can modify the current plan. Some updates are also sent back to CASPER which can adjust the long-term plan accordingly. The final system was demonstrated for generating an uplink sequence for contacting Mars Global Surveyor on a DSN 70m antenna.

The next step towards a tighter integration is shown in Figure 6.4. For this integration step there will

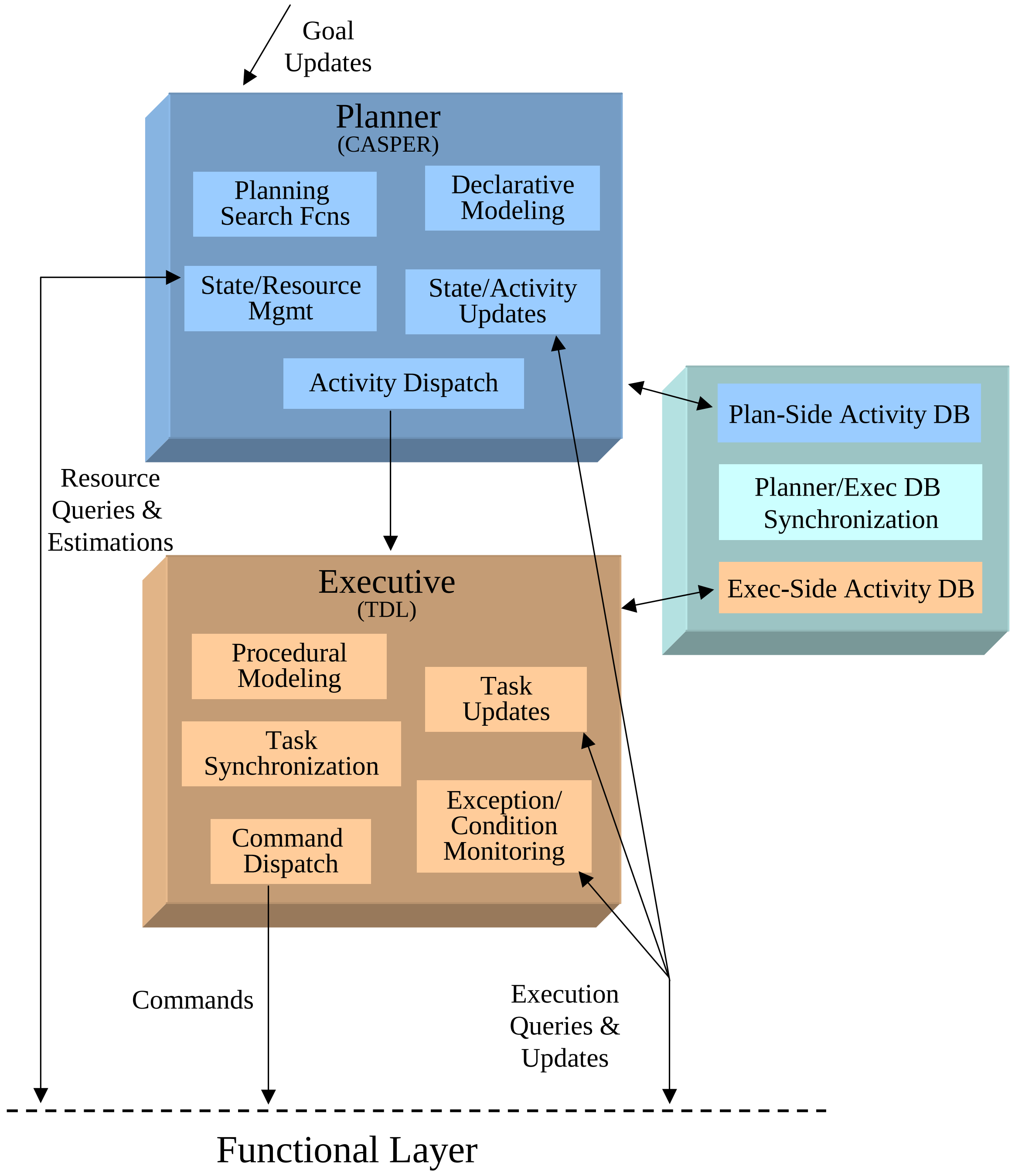


Figure 6.4: Tighter integration of CASPER planner and TDL executive. Two activity databases are tightly coupled together.

exist two tightly-coupled Activity Databases that hold the current plan for both the planner and the executive. Both representation styles will still be maintained in these two databases however the databases will be synchronized so as to contain the same information. Thus, if the planner makes a change in its database, this change will also be reflected in the executive database, and vice versa. In this framework goals will be placed directly on the database timelines where they can be expanded by either the planner or executive. The executive will still be the main interface to the Functional Layer and will be responsible for dispatching commands to Functional Layer capabilities. A separate State Determination module will be responsible for updating activities and state and resource timelines in the two databases with current information from the functional layer. The planner will be allowed to query the Functional Layer for resource estimations queries; this capability will be further discussed in Section 6.7.2.

6.6 Proposed CLARAty Decision Layer Framework

The future integration design for the planner and executive is shown in Figure 6.5. In this framework there is only one activity database and thus one plan and model representation that is utilized by both planning and executive functionality.[2] A library of these functions exists that can refine the current and future plan, which includes capabilities for declarative and procedural expansion, activity synchronization, state and resource management, exception handling. This library will also contain sophisticated planning and scheduling search methods. Methods can be included for both constructive scheduling (e.g., backtracking) and repair-based scheduling (e.g., iterative repair). The planning/scheduling method used for a particular architecture instantiation can be chosen based on items such as the application relevance or user preference. All planning and executive capabilities can operate on different levels of abstraction and on different timescales.

There are two separate modules that handle communication with the Functional Layer. The State Determination module handles activity, state and resource updates, and monitoring for new exceptions. This module must determine what timelines and activities should be modified and when these modifications should occur to avoid any inconsistencies during planning. Most updates will be sent periodically by the Functional Layer. Other updates may be sent asynchronously only when corresponding state or resource values significantly change or in response to queries from the Decision Layer. The State Determination module also handles resource queries and the resulting estimations. Resource queries are instigated when necessary based on new activity instantiations. This module will need to formulate the query and possibly determine whether a simple or extensive analysis should be performed to answer the query. Once an estimate is returned, the appropriate activity parameters must be updated. The Activity Dispatch module acts as a timeline runner for sending appropriate commands to the Functional Layer. This module will need to dispatch activities to the appropriate Functional Layer object based on their appointed start time and/or the value of other relevant preconditions. These interfaces are further discussed in the next few sections.

6.7 Interface to Functional Layer

The Decision Layer accesses the Functional Layer at a flexible point we call “The Line.” Two main things occur at this interface, which is graphically depicted in Figure 6.6. First, this is where the Decision Layer maps low-level plan activities to the built-in capabilities of the Functional Layer. Second, the Decision Layer receives information updates from the Functional Layer for items such as activity failure or success, activity duration, resource levels, state information, and exception notification.

[2] This representation language has not yet been completely defined and will be the subject of future research. However it will be a combination of declarative and procedural structures. It will also support flexible timepoints for activity temporal constraints [62].

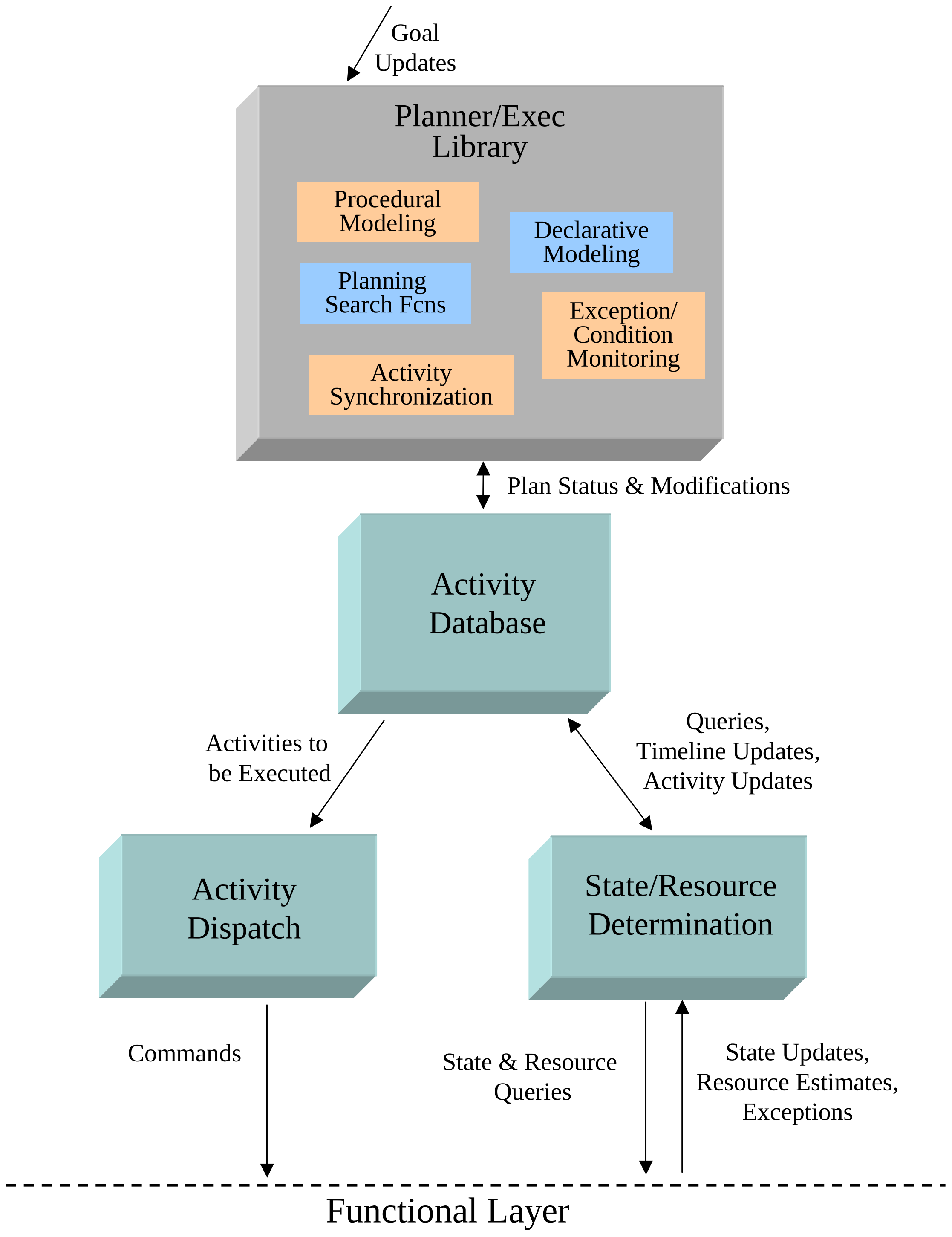

Goal Updates
Planner/Exec Library
Procedural Modeling
Declarative Modeling
Planning Search Fcns
Exception/ Condition Monitoring
Activity Synchronization
Plan Status & Modifications
Activity Database
Activities to be Executed
Queries, Timeline Updates, Activity Updates
Activity Dispatch
State/Resource Determination
Commands
State & Resource Queries
State Updates, Resource Estimates, Exceptions
Functional Layer

Figure 6.5: Future integration of planning and executive. Planner and executive use same activity database and data representation.

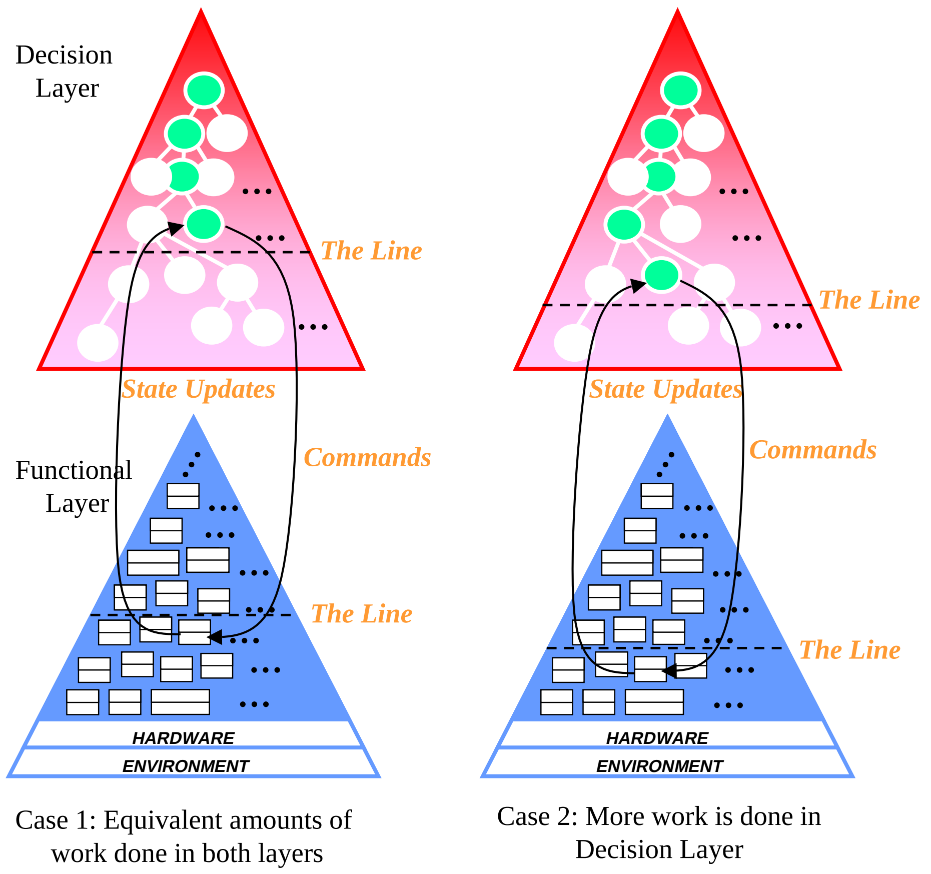


Figure 6.6: “The Line” is where the Decision Layer accesses the Functional Layer. This line is considered floating and can be adjusted depending on the application.

The position of “The Line” will be designated within the rover model used by the Decision Layer. This model is what defines for the Decision Layer the available activities, resource and states and any relevant constraints on these entities. The position of “The Line” will be reflected within this model before each architecture instantiation, and will remain static throughout the use of that instantiation. Please note that “The Line” may not reside at the lowest level of activity abstraction contained within a model. One model could be used for a number of different architecture instantiations, and “The Line” may exist in different places for each instantiation. In one case, “The Line” may exist at a very high level where any activities contained in the model that are located below “The Line” would not be accessed by the Decision Layer for that instantiation. But for another case the user may move the “The Line” to a lower level of abstraction, enabling the Decision Layer to access more low-level activities. The flexibility of “The Line”’ is further discussed in the next section.

6.7.1 Floating Line

In different situations, “The Line” may need to be moved so that the Decision Layer can access the Functional Layer at different levels of granularity. The position of the Line also affects what layer has control over certain decision-making functions. In Figure 1.5, the Line has been selected to reside just above the lowest level of objects in the Functional Layer. It is possible that the Line exists at the absolute base of the red triangle, or much closer to its peak.[3] In some cases, it might be more appropriate for the Decision Layer to break down tasks to a very detailed level and dispatch low-level commands to the Functional Layer where they are directly carried out. This enables complicated functionality to not always be built-in to the Functional Layer. For instance, you might have an uncommon but complicated science procedure that needs to be performed and it is not worthwhile to create a separate Functional Layer object for this procedure since it happens infrequently. This type of functionality could be more easily added to the Decision Layer model, which will be allowed in our framework. A flexible line also allows different levels of plan optimization to be performed. If global optimization of certain low-level activities is very important, than it might be beneficial to allow the Decision Layer to plan at a more detailed level where this optimization can be more easily realized.

In other cases, the Decision Layer may do very little refinement of tasks. Instead they may be sent directly to the Functional Layer where they must be broken down in a tight control loop or through some specific piece of real-time code. If all capabilities at a certain level of abstraction and below already exist in the Functional Layer, then there may be no need to lower the Line below these capabilities (unless, as mentioned above, there are certain optimization concerns that can only be handled by the Decision Layer). Furthermore, in some cases, the user of the architecture may be just more comfortable giving tighter control to one layer or to a particular module. Therefore, we intend for this interface line to be flexible. In this design, goal elaboration in the Decision Layer can be to different layers of granularity, depending on the domain application, robotic platform, or the user. The decision of where the line resides will initially be user controlled and will be set prior to a particular architecture instantiation. However, in future versions of this architecture we hope to explore having the line set and/or moved dynamically based on system performance.

Allowing this interface line to float may cause some duplication of functionality between the Decision Layer and Functional Layer. For instance, there may be similar information encoded in both Layers in order to break down a particular task since different instantiations of the architecture may give control of that task’s refinement to different layers. However, first, we feel this duplication is worth the flexibility enabled through this approach. Second, this information may be represented differently in the Decision Layer vs. the Functional Layer, which provides even more flexibility to the user and allows different metrics to be

[3]However, it is important to note that it must reside above the actual computer hardware. That is, no Decision Layer software accesses hardware directly. Hardware is controlled only through accessing objects in the Functional Layer.

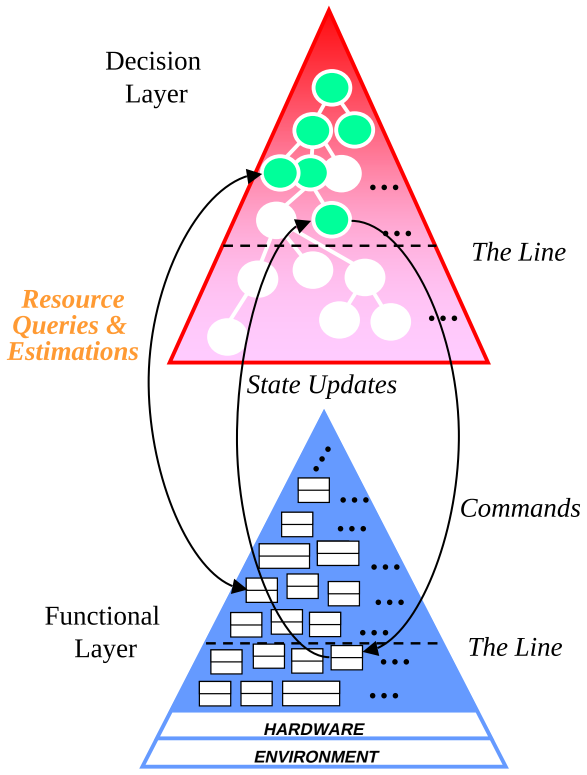


Figure 6.7: The Decision Layer can query the Functional Layer for resource usage predictions.

emphasized (e.g., efficiency vs. global resource utilization). One of the goals of this architecture is to use it for a wide range of robotic applications and thus the architecture will need to be tailored for each application.

6.7.2 Resource Prediction

The Decision Layer can also access the Functional Layer to request resource estimations. Information on expected resource usage is kept in the Functional Layer in the relevant objects for each resource. The Decision Layer can query the Functional Layer for resource usage predictions at varying degrees of resolution. This capability has already been discussed to some degree in Sections 1.3.6 and 4.7, and is graphically depicted in Figure 6.7. Examples of resource queries are how much battery power is required by a mast operation, how much memory storage is needed to contain data from a particular science operation, or how much time is required for a traversal between two locations. Resource queries may need to contain the type of activity requiring the resource, other relevant activity parameters (such as what time the activity is being performed), and other plan information that could impact that resource, such as other activities scheduled in that time window.

Resource queries can also be at different levels of resolution. The Decision Layer may request only a simple resource estimation if the activity using the resource is non-critical, far in the future, or if the planning window is very short (and thus there is not enough time for a detailed estimation). Conversely, the Decision Layer may request a very detailed resource estimation if the activity using the resource is for a critical or high-priority goal, or if the resource itself is difficult to predict accurately or its value often changes. However, since more detailed resource estimations may require additional time and computational resource to resolve there will be restrictions on when they can be performed. In some situations, the Functional Layer may have to replace a detailed query with a simple query if the needed computation power and time are not

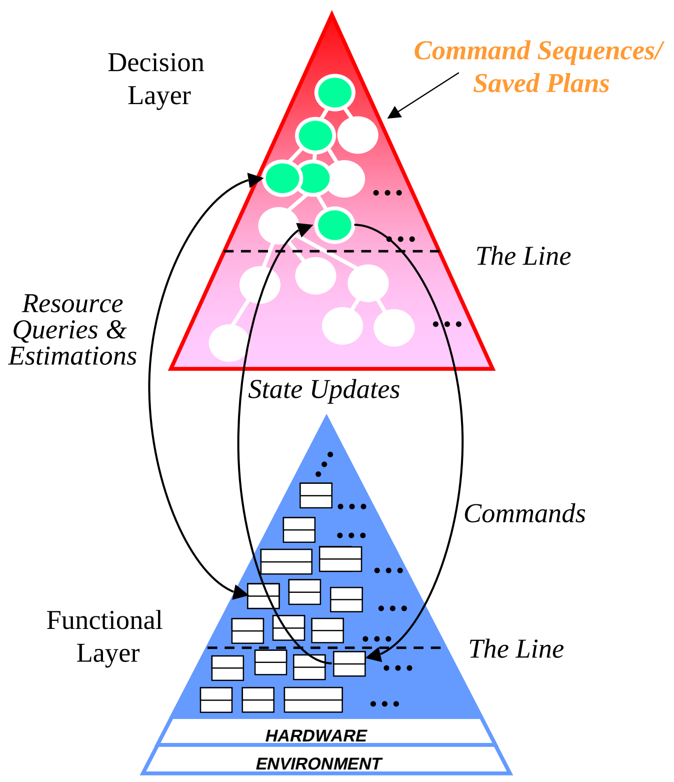


Figure 6.8: Command sequences and/or saved plans can be directly input to the Decision Layer.

available. Furthermore, the Decision Layer will need to know what Functional Layer objects hold estimators for what resources. This information is currently intended to be part of the Decision Layer model.[4]

Since resource queries can vary in requested detail, the returned estimations can correspondingly be at different levels of resolution. For a simple resource query, the returned estimation may be a simple scalar value or possibly a vector of values. For a more detailed resource analysis, the returned estimation may be represented by a continuous function over a certain time period. Since some Decision Layer systems may only be able to handle certain types of resource values (e.g., some planners may have no method of handling continuous-valued estimations), the Decision Layer instantiation may also dictate what type of queries are performed.

6.7.3 Direct Commanding and Saved Plans

As shown in Figure 6.8, the CLARAty architecture can support a *direct commanding* mode of operations if desired. This is the current mode of operations for the majority of NASA missions, and was the method used for commanding the Sojourner rover during the Mars Pathfinder mission [59]. In this mode, command sequences can be submitted to the Decision Layer directly from ground operations. No further elaboration will be performed on the sequences. Instead, they will be simply dispatched for execution to the Functional Layer.

In addition, it will be possible to save plans or sub-plans that have been created in advance or used for previous operations and then recall these plans into the Decision Layer at future times. Plans can be loaded

[4]One piece of future research is how to extract this type of information automatically be examining the Functional Layer implementation.

in the planner system or executive for further modification or could be directly executed. When using saved plans, the Decision Layer can still monitor activity, state and resource information and modify sequences when appropriate or desired.

6.8 Relation to MDS

As mentioned in Chapter 1, a new spacecraft software architecture project is ongoing at JPL. The Mission Data System (MDS) architecture [28] is a state-based object-oriented architecture that encourages a goal-based approach to commanding, as opposed to the traditional direct-commanding approach use to control spacecraft. This methodology opens the door for autonomy software to be easily used in missions that utilize this architecture. The effort of the CLARAty architecture is similar in spirit to MDS, but we have focused on a robotic architecture as opposed to a more general space flight and ground control architecture. However, we intend for our architecture to be compliant with the MDS effort and leverage MDS software elements. The rest of this section explains how to perform Decision Layer activities in MDS and discusses how our approach closely ties into the MDS architecture.

MDS utilizes a goal-based interface, similar to the interface presented for the CLARAty Decision Layer. MDS inputs goals, which are basically constraints on state, and then creates a sequence to achieve those goals using planning, scheduling, or other forms of elaboration and task refinement. Each state variable in MDS has a dedicated module that is responsible for achieving goals placed on that variable. These modules are titled Goal Achieving Modules (GAMs). Upon receipt of a goal, a GAM must decide if the goal can be achieved. The GAM then decides upon one or more actions to achieve the goal, such as dispatching commands to low-level controllers, rescinding conflicting goals, or subgoaling to other GAMS. This decision process is known as *elaboration*, and can be defined as the decision process of building and modifying goal constraint networks.[5] The basic elaboration process is shown in Figure 6.9. In MDS (and CLARAty) this constraint network is called a *goal net.* GAMs can modify the goal net in a number of ways including adding new subgoals, removing current goals, and adding new timepoints and temporal constraints. Executable (or low-level) goals in the goal net result in commands being dispatched to low-level controllers.

One important form of elaboration in MDS is AI planning. A goal of the MDS architecture effort has been to ensure that an AI planner can be integrated into the resulting framework [51]. Currently the planner being used for initial integration is the CASPER planning system, which performs soft real-time planning and re-planning. This type of planner is integrated into MDS by having it replace certain modules as shown in Figure 6.9. Conceptually, CASPER acts as a stand-in for certain state-variables and GAMs. Other GAMS are unaware that a planner has replaced certain parts of the architecture. The planner can accept goals for the GAMs it has replaced and can submit new subgoals to other GAMs if the planner cannot achieve them itself.

The selection of GAMs to be replaced by the planner is up to the system designer. In general, the more state-variables and GAMs that a planner can reason about, the more globally optimal the resulting plans will be since the planner will have more overall knowledge about the states, resources, and activities under its domain. However, the MDS GAM-based architecture is designed to easily allow for distributed elaboration, where goal elaboration is distributed among different GAMS, which could provide for more local optimality. These GAMS could be individual planners, scripts, or other programs designed for the specific purpose of achieving a particular goal. Sometimes it may be more appropriate for a state-variable to have a particular script or program that handles it and sometimes it may be more appropriate to have a planner that handles a set of related state-variables. Both of these options are allowed in the MDS architecture.

[5]We have also adopted this term to represent the refinement of goals in the CLARAty Decision Layer.

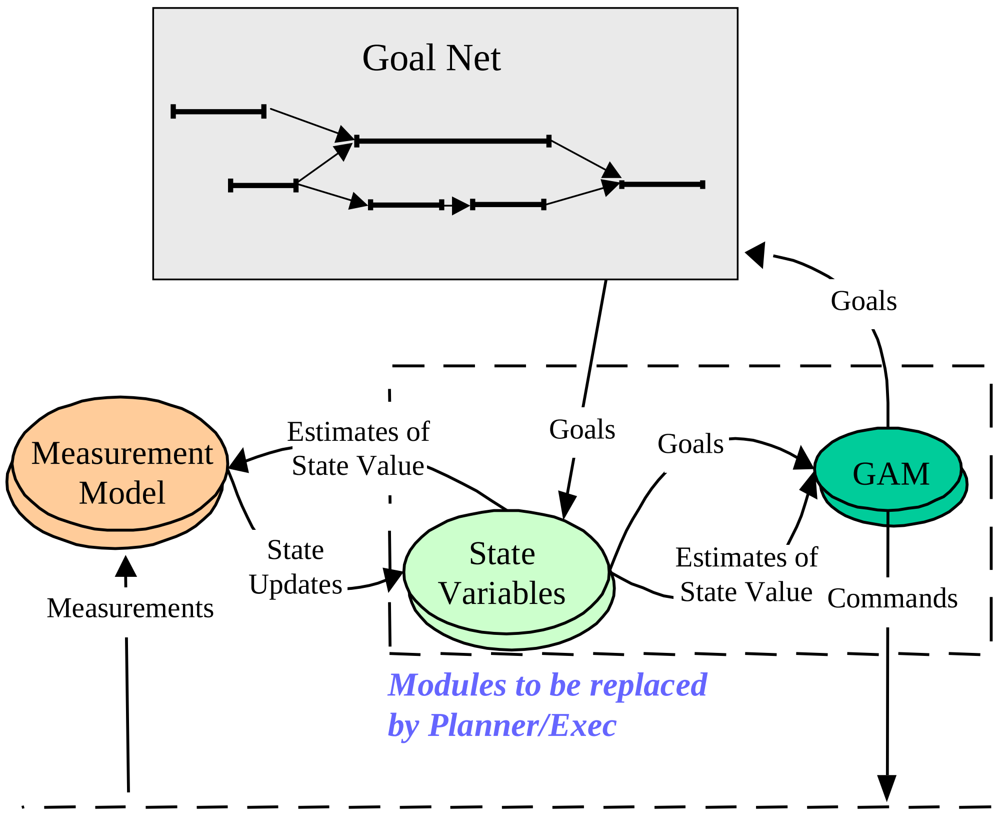


Figure 6.9: Goal elaboration in the MDS architecture. Modules that could be replaced by a Planner and/or Executive are shown.

When compared to the CLARAty Decision Layer, the framework shown in Figure 6.9 has many similarities but some differences. Both architectures employ similar planning and scheduling techniques to perform some goal elaboration and to generate schedules. Both maintain state through the use of activities scheduled on timelines. And, both architectures have also implemented the concept of “The Line” which separates high-level goal-oriented functions from more low-level control functions. For the CLARAty architecture we have focused on primarily using a centralized approach to planning where one planner has control of all state-variables and performs all elaboration functions. MDS, conversely, is more focused on a distributed GAM structure where a planner may only replace a certain subset of GAMS. However, for future designs of CLARAty we do intend to consider the ability to allow planning to be distributed across different modules if warranted by the application or desired by the user.

MDS does provides some simple executive functionality in the above the Line portion of the architecture. This functionality includes dispatching activities to be executed by a controller at the appropriate time, checking for activity preconditions before execution, and performing some procedural decomposition when building a schedule. Other executive functionality, such as closed-loop control, execution monitoring, and other reactive behavior based on current conditions (e.g., if/thens), is expected to be handled by low-level controllers below “The Line.” CLARAty, on the other hand, will have the ability to handle all types of executive functionality above the line and will enable the user to decide where certain operations should be performed in the architecture.

Part IV

Appendices

Appendix A

Examples

This chapter presents some simple examples intended to illustrate specific features of the CLARAty architecture. The intent is to ground some of the previous description by showing how to build specific types of systems within this framework. The examples that follow are: elaboration to different levels of granularity, ground sequencing versus on-board planning, resource estimates to different precision, alternate control techniques, and modular addition of hardware to the system.

A.1 Elaboration to Different Levels of Granularity

As described previously, the overlapping Functional and Decision Layers of CLARAty allow greater or lesser granularity in each layer. This feature can be used by the system designer to address different modes of operation. Figure A.1 shows a simplified comparison of greater of lesser granularity in the Decision Layer, and the corresponding access points to components of the Functional Layer. In this example, a simple mission goal is broken down into goals for navigating to a science target and obtaining a science measurement there. In Case 1, the navigation goal is achieved by accessing latent functionality built into the rover object. While this object may access others such as a path planner or locomotor subsystems, these details are hidden from the Decision Layer. Figure A.2 shows the physical situation in the left picture, where the rover uses its Functional Layer to navigate to the specified target.

Alternatively, the Decision Layer may elaborate the navigation goal into subgoals that may be implemented as a series of intermediate way-points leading to the science target. The right sides of Figures A.1 and A.2 show representations for the software interaction and rover activity in this scenario. The disadvantage of this implementation is that the Decision Layer must do more elaboration of the goals, concern itself with finer levels of granularity and the accompanying state and resource details, and include the local intermediate traverse plans (typically provided by query of the path planner). The advantage of this implementation is the availability of more detailed information to the global planning capability built into the Decision Layer. Therefore, other needed activities may be considered in addition to navigation, and complete system resource usage predictions may be used to schedule activities at the same time, or interspersed with navigation.

Comparison of the two strategies may also be shown on the activities time-line for the system, shown in Figure A.3 Whereas, the first case would be completely represented by the second time-line from the top, the second case is illustrated by the bottom time-line. In this latter case, an ‘engineering’ activity has autonomously been placed between the series of navigation ‘goto’ operations, typically to make performance more optimal.

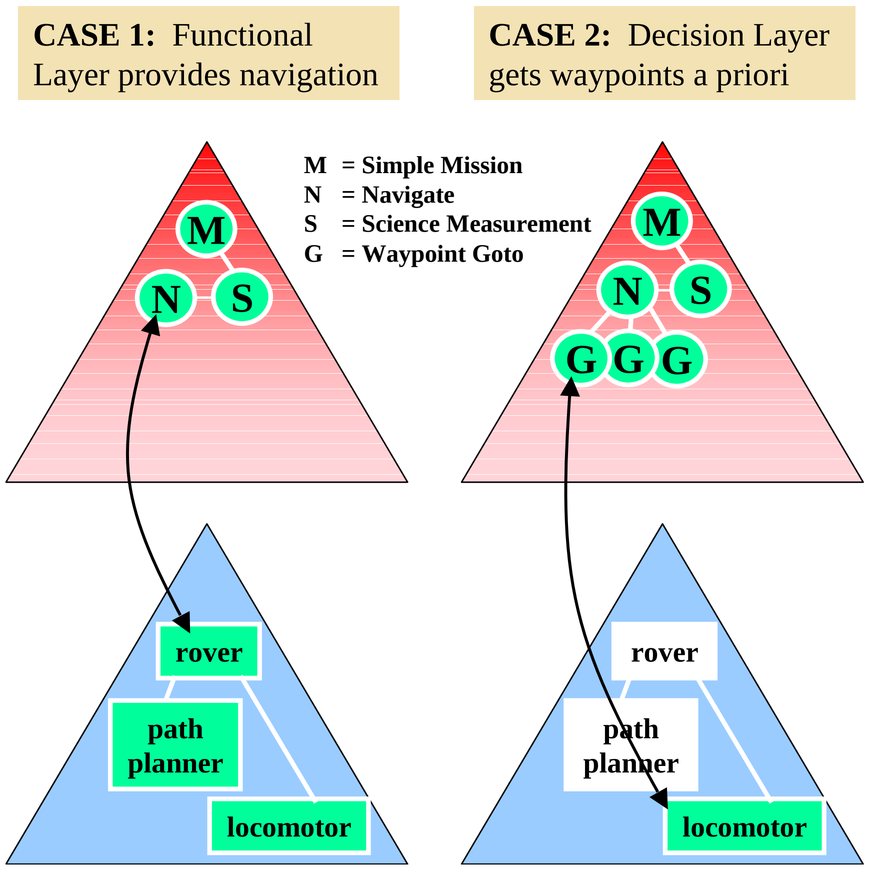


Figure A.1: Greater or lesser granularity or elaboration and access of the Functional Layer.

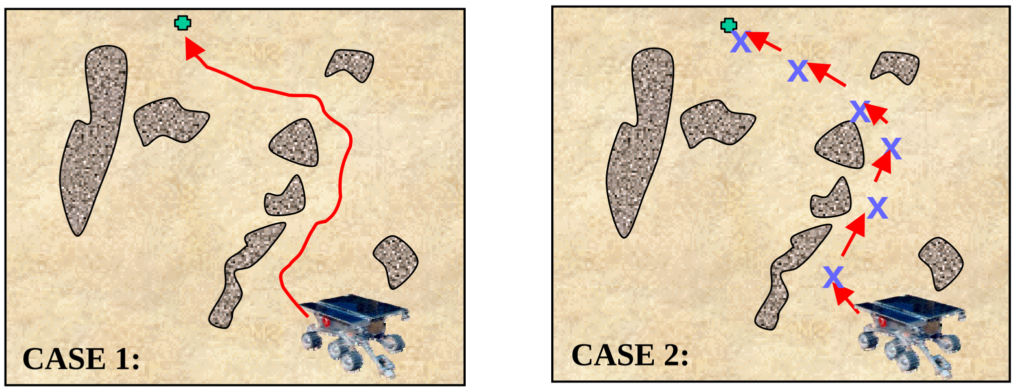


Figure A.2: Rover actions when controlled at greater or lesser granularity.

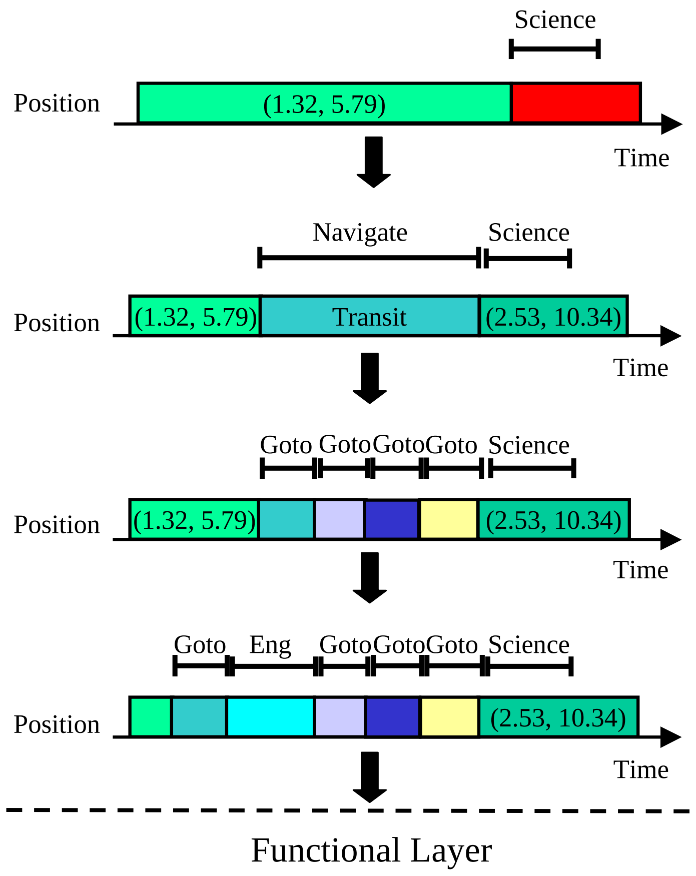


Figure A.3: More task resolution makes scheduling more flexible and optimal. For instance, in this time-line the planner inserts an engineering task during navigation.

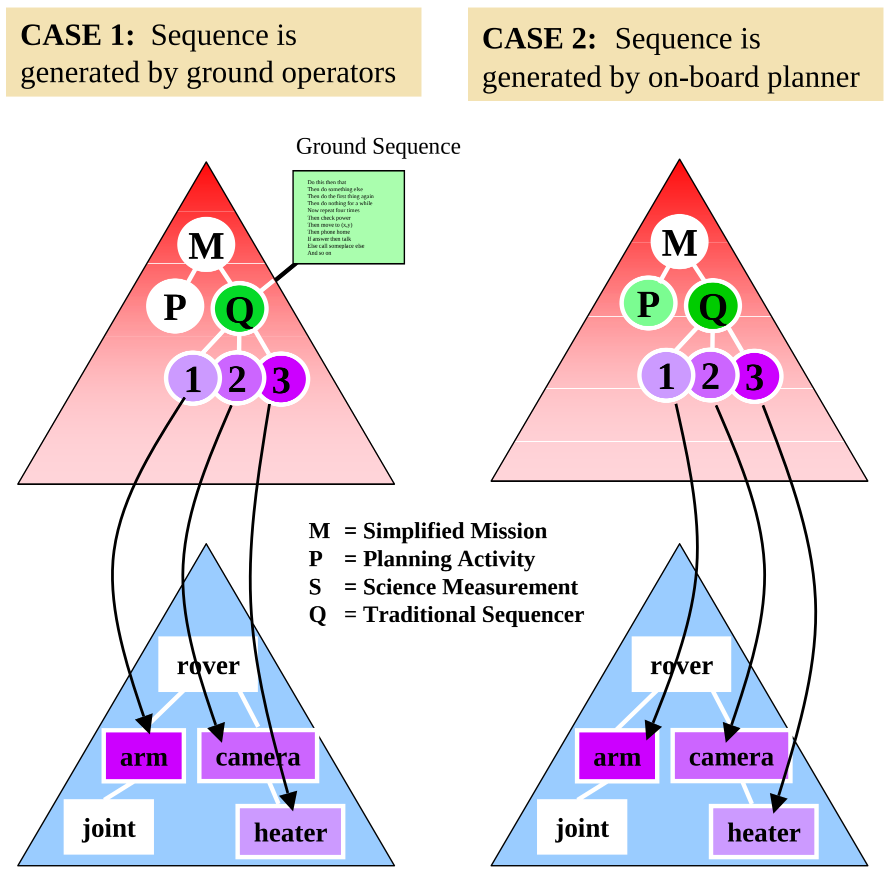


Figure A.4: Implementing traditional sequences within CLARAty.

A.2 Ground Sequencing versus On-board Planning

Another mode of system operation that CLARAty will support is traditional sequence parsing and command execution. While the primary purpose of the Decision Layer is to provide autonomy to the system, it may be desirable at times to bypass the majority of these capabilities. This could be due to a number of factors including anomalous conditions, operator choice, testing, etc.

Figure A.4 compares the two modes of operation. On the left is shown a traditional ground sequence which is created by an operator (on the ground, in the case of spacecraft operations). This script is parsed by a sequencer, and each specified operation is achieved by accessing the capabilities of the Functional Layer. Such operation is largely open-loop to the Decision Layer, bypassing its ability to monitor system state and adjust accordingly. Instead, reports to the operator will force the creation of new sequences to accommodate unexpected performance during the sequence execution.

Alternatively, the right side of Figure A.4 shows the Functional Layer activities, but in this case they have been planned and scheduled by the Decision Layer, working autonomously to achieve the mission goal. Implied in this mode of operation, is the monitoring of state and execution status by the Decision Layer, and replanning when necessary. In this way, the operator is removed from the goal achieving process, and forced to interface with the layer by goal specification at higher levels of granularity.

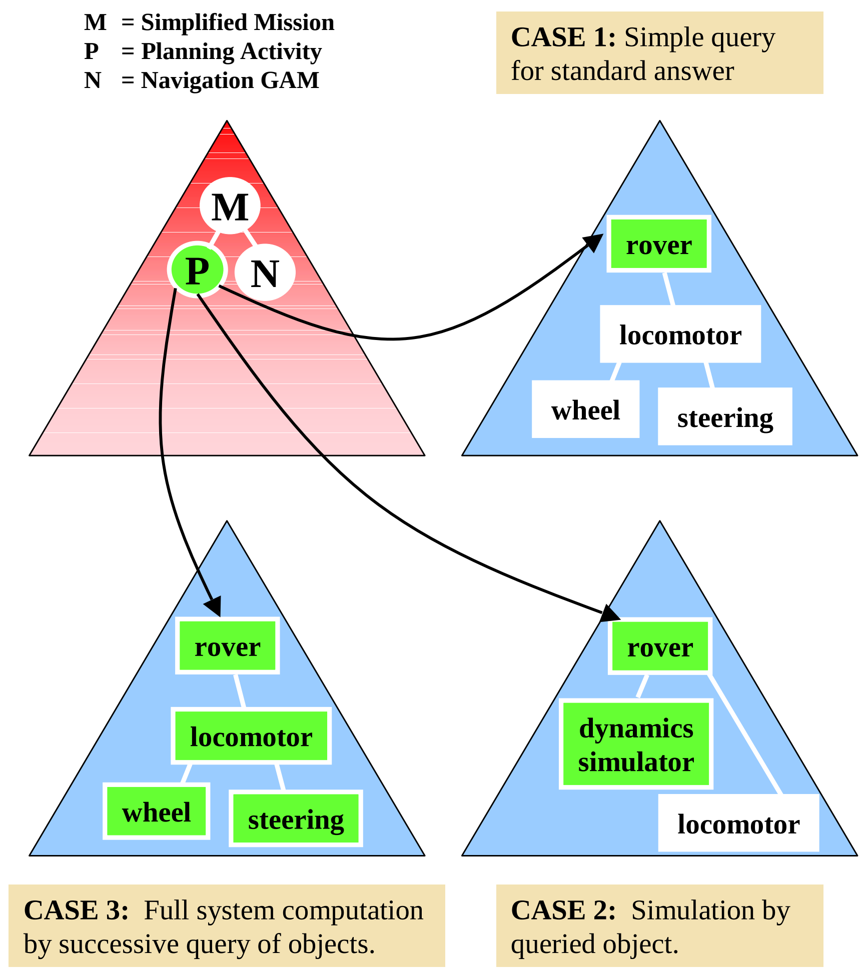


Figure A.5: Obtaining a resource estimate to different levels of precision.

A.3 Resource Estimates to Different Precision

When the Decision Layer is scheduling operations, it requires estimates of resource use during these operations. CLARAty is designed such that resource estimates are obtained by the appropriate objects of the Functional Layer, as shown in Figure A.5. In this figure, three different versions of resource requests for rover traversal are shown. Each resource request is parameterized by a level of requested precision, where the allowed precision levels are specified as part of the interface to the object.

At a minimum the resource estimate provided by the Functional Layer is a single scalar value which is a simple estimate of average resource use based on prior use or system specifications. Such a value may be hard-coded, or obtained by monitoring and averaging of past performance. It is provided directly by the Functional Layer object queried, and for the purposes of planning and scheduling it is assumed to be a constant over the full duration of the activity. In the example of Figure A.5, Case 1 shows the planner obtaining a scalar power estimate for a rover traversal directly from the rover object.

If resource availability is limited, or if other reasons require planning with more precision, then a more detailed estimate of power usage is desired. In this case, two improvements can be made: recursive subsystem inquiries, and vectorized resource usage specification. The former is necessary to improve the estimate, and the latter to capture the value of the improved estimate. Case 2 in Figure A.5 shows the rover object

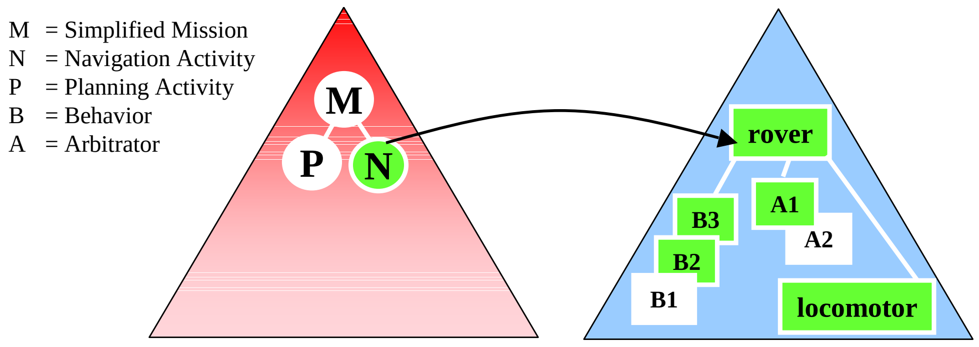


Figure A.6: Decision Layer access of Functional Layer modes of control implemented as behaviors.

obtaining a more precise estimate of resource usage by accessing a dynamics simulation of the rover. Alternatively, Case 3 in the same figure shows the rover object recursively querying subordinate objects to obtain a more precise resource usage estimate. In this example, the locomotor object obtains power usage estimates from the wheel and steering controllers, which potentially use internal wear models to increase the accuracy of their predictions.

A.4 Alternate Control Techniques

To be flexible for applications as well as research, it is important that CLARAty can incorporate alternate control techniques. One important area of alternate control in robotics is Behavior Control. Figure A.6 illustrates how behaviors and arbitrators may be included in the Functional Layer as subordinate objects that are aggregated into relevant objects such as the rover. In this way, they can be activated by the rover at its discretion, to accomplish objectives given to it from the Decision Layer. This structure is essentially the same as with other controller objects that may be aggregated by the rover, but implicitly includes the idea of multiple object controllers working at once, with arbitration amongst them. It also allows for the activation of behaviors and arbitrators by the Decision Layer directly, but the selection of these combinations by a planner is not currently common practice. However the architecture does allow it, and the associated research necessary to accomplish it.

A.5 Changing Hardware Capabilities of the System

Modular, object-oriented programming, portends the use of complementary, modular, robotic hardware. In the multi-wheeled robots discussed thus far, there is already the use of the framework for system components such as wheels. In this case, one wheel object is instanced six times for control of the six wheels on a typical planetary rover. While this is an obvious use, less obvious is the implicit extensibility of the system that is strictly object oriented. Physical subsystems such as stereo pairs or arms may be added, and the controlling software is immediately ready for instancing. This principle extends to complete mobile robots, where multiple instances may be made for cooperating teams acting as a unit. Further, it makes tractable the software and control problems of a very large set of strictly modular, dynamically reconfigurable robot building blocks.

To illustrate this point, Figure A.7 shows a simple example of extending an existing system by adding a second arm to it. Making a second instance of the arm software is easily accomplished, but opens up two possibilities for access of this new object. First, if the Decision Layer accesses the system only at higher levels of granularity (for instance at the rover object level). In this case, changes to the rover object may be

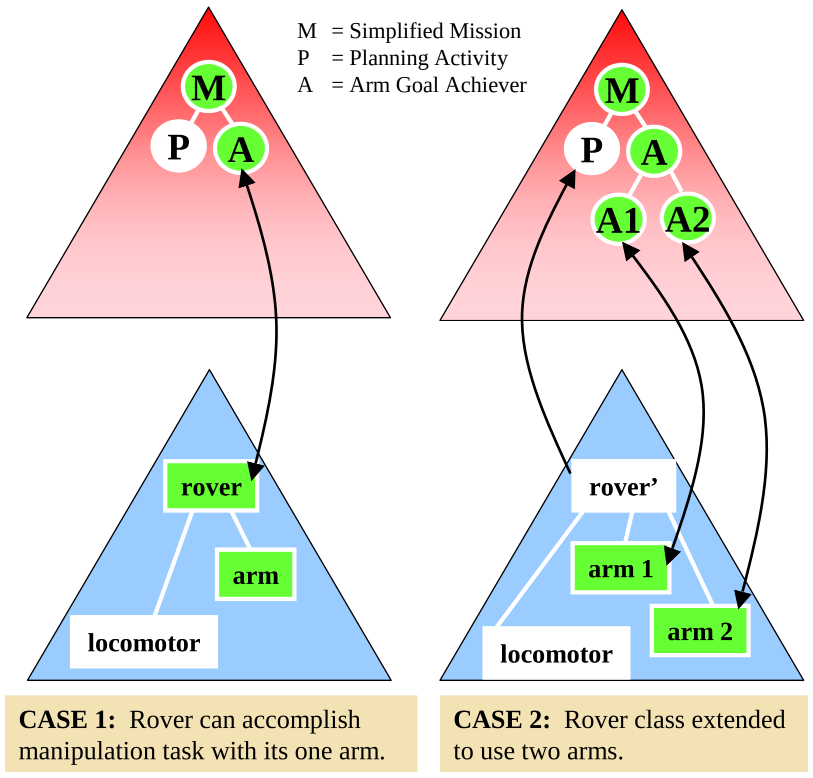


Figure A.7: System hardware changes are easier to accommodate by multiple instantiations of object classes, and compartmentalization of control access in parent object.

needed to generalized it for two arms. (These changes are denoted by the prime attached to the rover object name in the right side of the figure.) Alternatively, if the Decision Layer is directly accessing lower levels of granularity, then the addition of the second arm must be made known to it. Currently methods require this be done through manual changes of Decision Layer model files, but automatic updating is planned for the future.

One example is shown in Figure A.7, where only the rover is queried during planning for resource usage predictions, but the arms are commanded separately during execution. This scenario might be useful for planning using gross power estimates, but executing specialized coordination of arm activity that is not provided by the rover object.

Appendix B

Status

B.1 Leveraging Legacy Software

The implementation of the CLARAty architecture is only in its early stages, but some important progress has been made about tool selection and legacy software leveraging.

Functional Layer

UML description: Initial stage of system description being completed; extensions starting.

Basic Functionality: Rocky 7 mobility, manipulation, and visual processing reuse from PDM task.

Hiding Hardware Details: Motor classes modified to handle Rocky 7 (DIO, LM629) and Rocky 8 (I2C, PIC, HCTL1100) motor control hardware.

Enhancements: New functionality added to handle continuous driving/steering.

Navigation: Integration of Morphin/D* in progress as part of IS program effort.

Decision Layer

First Planner: Tests have begun with CASPER planner and its mini-exec controlling Rocky 7.

First Executive: Plans to leverage CLEaR task effort for integration of CASPER with TDL.

Resource Management: Initial investigation of APIs for resource information from Functional Layer.

B.2 Summary of Experimental Results

We have used several components presented here to control various robotic platforms including: Rocky 7 rover and a PDM rover mockup platform. The Rocky 7 rover has a 3U VME backplane with a 60 MHz 68060 processor with on-board Ethernet, two PC104 Imagenation frame-grabbers, a PC104 to VME board, a VPAR10 digital I/O board, and a VADC20 analog I/O board. The main processor runs a VxWorks 5.3 real-time operating system. Each actuator (DC brushed) is controlled by a separate micro-controller (LM629). The controllers are connected to the CPU through a multiplexed shared bus using 8 bit parallel port lines of the digital I/O board. The on-board processor communicates with an external host via wireless ethernet. The Rocky 7 rover is a six-wheel drive rover with two-front-wheel steering. It uses a rocker-bogie suspension mechanism. The arm has four degrees of freedom (DOF), two of which control the end effector scoops. In effect, it is only a 2-DOF point device. The mast has 3-DOFs, which include an elbow joint.

The second PDM rover mockup used a cPCI backplane with a 300 MHz Pentium processor, a separate CP340 Ethernet card, two cPCI PX610 framegrabbers, a Sensoray S720 digital I/O board. The motor control used the S720 board with LM629 micro-controllers. It has a 5-DOF arm with an elbow and a wrist joint. The mast is also a 4-DOF manipulator with an elbow and a head (wrist) joint. This mockup did not have mobility.

Components at different levels of abstraction were shared between these two systems. Using Rocky 7, we demonstrated autonomous multiple sample acquisitions of selected targets and instrument pointing on designated targets. We also demonstrated porting of navigation algorithms developed at Carnegie Mellon University [74]. Using these components with different specialization, we demonstrated the vision-based acquisition of selected samples using the PDM mockup. Parts of these components are currently being integrated to operate on Rocky 8, which has a completely different hardware design. It uses a distributed architecture based on widget board motor controllers and I/O packages connected via an I2C bus.

Appendix C

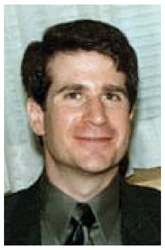
Acknowledgements

As discussed, some of the concepts in this report leverage those developed by the Mission Data System team at JPL. We would like to thank them for their continued interaction on architecture design issues.

The research described in this report was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

Appendix D

Biography



Richard Volpe, Ph.D., is the Principal Investigator for the Long Range Science Rover Research Team. His research interests include real-time sensor-based control, robot design, software architectures, path planning, and computer vision.

Richard received his M.S. (1986) and Ph.D. (1990) in Applied Physics from Carnegie Mellon University, where he was a US Air Force Laboratory Graduate Fellow. His thesis research concentrated on real-time force and impact control of robotic manipulators. Since December 1990, he has been at the Jet Propulsion Laboratory, California Institute of Technology, where he is a Senior Member of the Technical Staff. Until 1994, he was a member of the Remote Surface Inspection Project, investigating sensor-based control technology for telerobotic inspection of the International Space Station. Starting in 1994, he led the development of Rocky 7, a next generation mobile robot prototype for extended-traverse sampling missions on Mars. In 1997, he received a NASA Exceptional Achievement Award for this work, which has led to the design concepts for the 2003 Mars rover mission.

Issa A.D. Nesnas, Ph.D., is the cognizant engineer for the Architecture and Autonomy Research task. His research interests include software and hardware architectures for robotic systems, autonomous sensor-based coordination, actuation and control, computer vision, object-oriented design, and artificial intelligence. Issa received a B.E. degree in Electrical Engineering from Manhattan College, NY, in 1991. He earned the M.S. and Ph.D. degrees in robotics from the Mechanical Engineering Department at the University of Notre Dame, IN, in 1993 and 1995 respectively. In 1995, he joined Adept Technology Inc. as a senior project engineer inventing and implementing several new technologies for high-speed vision-based robotic applications and holds a patent for the Impulse-based flexible parts feeder. He also participated in the NCMS Consortium working with industry leaders in factory automation such as Ford, GM, Delco Electronics, and Cummins Engine designing hardware and software standards for robotic assembly cells. He has joined NASA at the Jet Propulsion Laboratory as a member of technical staff in 1997. At JPL he has worked on the Planetary Dexterous Manipulator project researching autonomous sensor-based manipulation for rovers. In addition to his duties on the Architecture and Autonomy task, Issa is also working on a flight project for autonomous sensor-based landing on Mars. He has received several Notable Organizational Value Added (NOVA) Awards and an Exceptional Achievement Award for his work at JPL. Issa is a member of Eta Kappa Nu and Tau Beta Pi National Honor Societies.

Tara Estlin, Ph.D., is a senior member of the Artificial Intelligence Group at the Jet Propulsion Laboratory in Pasadena, California where she performs research on planning and scheduling systems for rover automation and multi-rover coordination. Dr. Estlin has led several JPL efforts on automated rover-command generation and planning for distributed rovers, and is a team member of the JPL Long Range Science Rover team. She received a B.S. in computer science in 1992 from Tulane University, an M.S. in computer science in 1994 and a Ph.D. in computer science in 1997, both from the University of Texas at Austin. She has numerous publications in planning/scheduling and machine learning, including such high profile forums such as AAAI, IJCAI, and ICRA. Her current research interests are in the areas of planning, scheduling, and multi-agent systems.

Darren Mutz received the Bachelor of Science degree in Computer Science at the University of California, Santa Barbara in 1997. Current projects include Long Range Science Rover (LRSR), statistical hypothesis evaluation, and the Automated Planning and Scheduling Environment (ASPEN).

Richard Petras is a member of the Telerobotics Research and Applications Group at the Jet Propulsion Laboratory, California Institute of Technology. He is currently a member of the Long Range Science Rover (LRSR) Team, supporting the design of a robotic architecture for planetary rovers. Previous work at JPL involved design of the ORCAA architecture for the Rocky 7 research rover, development of algorithms for an articulated camera mast, and low level software for motor control, vision systems, and science instruments. Before coming to JPL Rich worked on Space Shuttle and Space Station avionics for IBM Federal Systems Division (later Loral Space Information Systems) implementing redundant hardware and software. Rich graduated from Drexel University in 1985 with a B.S. in Mechanical Engineering. He got his M.S. in Space Sciences from the University of Houston, Clear Lake in 1994.

Hari Das, received his ScD degree on Mechanical Engineering from MIT in 1989. He is a Senior Member of Technical Staff at JPL. His research interests are in the development and evaluation of robotic systems for biomedical and planetary exploration applications.

Bibliography

[1] ImageVision Library. Silicon Graphics, Inc., Mountain View, CA.

[2] Mobility Software. Real World Interface, a division of IRobot, Somerville, MA.

[3] Vector Signal Image Processing Library. Georgia Tech Research Institute, Georgia.

[4] R. Alami, R. Chautila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17(4), April 1998.

[5] R. Alami et al. An Archtecture for Autonomy. *International Journal of Robotics Research*, 17(4), April 1998.

[6] 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, MD, July 1987.

[7] James Albus. 4-D/RCS Reference Model Architecture for Unmanned Ground Vehicles. In *IEEE Internation Conference on Robotics and Automation*, San Francisco, April 24-27 2000.

[8] J. A. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, St. Paul, MN, July 1988.

[9] M. Austern. *Generic Programming and the Stl: Using and Extending the C++ Standard Template Library*. Addison-Wesley Professional Computing Series, Reading, MA, October 1998.

[10] P. Backes et al. Automated Planning and Scheduling for Planetary Rover Distributed Operations. In *IEEE Internation Conference on Robotics and Automation*, Detroit MI, May 1999.

[11] P. Backes, M. Long, and R. Steele. The Modular Telerobot Task Execution System for Space Telerobotics. In *IEEE Internation Conference on Robotics and Automation*, Atlanta Georgia, May 1993.

[12] J. Balaram. Kinematic State Estimation for a Mars Rover. *Robotica, Special Issue on Intelligent Autonomous Vehicles*, 18:251–262, 2000.

[13] J. Balaram and H. Stone. Automated Assembly in the JPL Telerobot Testbed. In *Intelligent Robotic Systems for Space Exploration*, Norwell, MA, 1992. Kluwer Academic Publishers.

[14] A. Bejczy. Robot Arm Dynamics and Control. Technical Memorandum 33-669, Jet Propulsion Laboratory, Pasadena, CA, February 1974.

[15] A. Bejczy and Z. Szakaly. An 8-D.O.F. Dual-Arm System for Advanced Teleoperation Performance Experiments. In *Fifth Annual Workshop on Space Operations Applications and Research (SOAR '91)*, Houston TX, July 9-11 1991.

[16] Doug Benard, G. Dorais, C. Fry, E. Gamble, B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, P. Nayak, B. Pell, K. Rajan, N. Rouquette, B. Smith, and B. Williams. Design of the remote agent experiment for spacecraft autonomy. In *Proceedings of the 1998 IEEE Aerospace Conference*, Aspen, CO, March 1998.

[17] D. Biesiadecki, J. Henriquez and A. Jain. A Reusable, Real-time Spacecraft Dynamic Simulator. In *6th Digital Avionics Systems Conference*, Irvine, CA, October 1997.

[18] R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence Research*, 9(1), 1997.

[19] J. Borrelly et al. The ORCCAD Architecture. *International Journal of Robotics Research*, 17(4), April 1998.

[20] S. Bozic. *Digital and Kalman Filtering: An Introduction to Discrete-Time Filtering and Optimum Linear Estimation.* John Wiley and Sons, September 1994.

[21] John Bresina, Keith Golden, David Smith, and Rich Washington. Increased flexibility and robustness of mars rovers. In *Proceedings of the 1999 International Symposium on Artficial Intelligence, Robotics and Automation for Space*, Noordwijk,The Netherlands, June 1999.

[22] R. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal on Robotics and Automation*, 2(1), March 1986.

[23] S. Chien, R. Knight, R. Sherwood, and G. Rabideau. Integrated Planning and Execution for Autonomous Spacecraft. In *IEEE Aerospace Conference*, Aspen CO, March 1999.

[24] Steve Chien, Russell Knight, Andre Stechert, Rob Sherwood, and Gregg Rabideau. Using iterative repair to improve responsiveness of planning and scheduling. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, Aspen, CO, April 2000.

[25] B. Douglass. *Real-Time UML — Developing Efficient Objects for Embedded Systems*. AWL, Reading, MA, December 1998.

[26] Brian Drabble, J. Dalton, and Austin Tate. Repairing plans on the fly. In *Proceedings of the First NASA International Workshop on Planning and Scheduling for Space*, Oxnard, CA, October 1997.

[27] D. Dvorak, Rasmussen R., G. Reeves, and A. Sacks. Software Architecture Themes in JPL's Mission Data System. In *IEEE Aerospace Conference*, Big Sky, Montana, March 2000.

[28] Daniel Dvorak, Robert Rasmussen, Glenn Reeves, and Allan Sacks. Software architecture themes in jpl's mission data system. In *Proceedings of the 2000 IEEE Aerospace Conference*, Aspen, CO, March 2000.

[29] K. Erol, D. Nau, and J. Hendler. UMCP: A sound and complete planning procedure for hierarchical task-network planning. In *Proceedings of the Second International Conference of AI Planning Systems*, Chicago, June 1994.

[30] T. Estlin, G. Rabideau, D. Mutz, and S. Chien. Using Continuous Planning Techniques to Coordinate Multiple Rovers. In *IJCAI Workshop on Scheduling and Planning*, Stockholm, Sweden, August 1999.

[31] Tara Estlin, Alexander Gray, Tobias Mann, Gregg Rabideau, Rebecca Castano, Steve Chien, and Eric Mjolsness. An integrated system for multi-rover scientific exploration. In *Proceedings of the Sixteenth National Conference on Ariticial Intelligence*, Orlando, FL, July 1999.

[32] Tara Estlin, Gregg Rabideau, Darren Mutz, and Steve Chien. Integrating planning and execution for multiple rover operations. In *Proceedings of the Second NASA International Workshop on Planning and Scheduling for Space*, San Francisco, CA, March 2000.

[33] R. Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, Department of Computer Science, 1989.

[34] R. James Firby. *Adaptive Execution in Dynamic Domains*. PhD thesis, Yale University, New Haven, CT, 1989.

[35] F. Fisher et al. A Planning Approach to Monitor and Control for Deep Space Communications. In *IEEE Aerospace Conference*, Big Sky MT, March 2000.

[36] Forest Fisher, Barbara Engelhardt, Tara Estlin, and Steve Chien. untitled. In *Submitted to the 2001 IEEE Conference on Robotics and Automation*, Seoul, Korea, May 2000.

[37] Forest Fisher, Russell Knight, Barbara Engelhardt, Steve Chien, and Niko Alejandre. A planning approach to monitor and control for deep space communications. In *Proceedings of the 2000 IEEE Aerospace Conference*, Aspen, CO, March 2000.

[38] Mark Fox. ISIS: A retrospective. In Monte Zweben and Mark Fox, editors, *Intelligent Scheduling*, pages 3–28. Morgan Kaufmann, San Francisco, CA, 1994.

[39] Alex Fukanaga, Gregg Rabideau, Steve Chien, and David Yan. Towards an application framework for automated planning and scheduling. In *Proceedings of the 1997 International Symposium on Artficial Intelligence, Robotics and Automation for Space*, Tokyo, Japan, July 1997.

[40] E. Gamma et al. *Design Patterns*. AWL, September 1999.

[41] E. Gat. On Three-Layer Architectures. In D. Kortenkamp, R. Bonnasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots*, Boston, MA, 1998. MIT Press.

[42] E. Gat et al. Behavior Control for Robotic Exploration of Planetary Surfaces. *IEEE Transactions on Robotics and Automation*, 10(4):490–503, 1994.

[43] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA, July 1992.

[44] Erann Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the 1997 IEEE Aerospace Conference*, 1997.

[45] Erann Gat. Three layer architectures. In D. Kortenkamp, R. Bonasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, Menlo Park, CA, 1998.

[46] M. Georgeoff and A. Lansky. Reactive reasoning and planning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, Seattle, WA, Jul 1987.

[47] D. Hanselman and B. Littlefield. *Mastering Matlab 5: A Comprehensive Tutorial and Reference*. Prentice Hall, NJ, December 1997.

[48] V. Hayward and R. Paul. Robot Manipulator Control Under Unix RCCL: A Robot Control "C" Library. *International Journal of Robotics Research*, 5(4), Winter 1986.

[49] http://www.intel.com/research/mrl/research/cvlib/. *Open Source Computer Vision Library.* Intel.

[50] A. Jonsson, P. Morris, N. Muscettola, K. Rajan, and B. Smith. Planning in interplanetary space: Theory and practice. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, Aspen, CO, April 2000.

[51] R. Knight, S. Chien, T. Starbird, K. Gostelow, and R. Keller. Integrating model-based artificial intelligence planning with procedural elaboration for onboard spacecraft. In*Proceedings of Space Ops 2000*, Toulouse, France, June 2000.

[52] R. Knight et al. Integrating Model-based Artificial Intelligence Planning with Procedural Elaboration for Onboard Spacecraft Autonomy. In *SpaceOps Conference*, Toulouse France, June 2000.

[53] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti. The Saphira architecture: A Design for Autonomy. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1):215–235, 1997.

[54] P. Laborie and M. Ghallab. Planning with shareable resource constraints. In*Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Canada, August 1995.

[55] S. Laubach. *Theory and Experiments in Autonomous Sensor-Based Motion Planning with Applications for Flight Planetary Microrovers*. PhD thesis, California Institute of Technology, May 1999.

[56] M. Maimone, I. Nesnas, and H. Das. Autonomous Vision-Based Manipulation from a Rover Platform. In *IEEE Symposium on Computational Intelligence in Robotics and Automation*, pages 351–356, Monterey, California, November 1999. http://robotics.jpl.nasa.gov/tasks/pdm/papers/cira99/.

[57] J. Matijevic et al. The Pathfinder Microrover. *Journal of Geophysical Research*, 102(E2):3989–4001, 1997.

[58] S. Minton and M. Johnston. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1988.

[59] Andrew Mishkin, Jack Morrison, Tam Nguyen, Henry Stone, Brian Cooper, and Brian Wilcox. Experiences with operations and autonomy of the mars pathfinder rover. In*Proceedings of the 1998 IEEE Aerospace Conference*, Aspen, CO, March 1998.

[60] Stewart Moorehead, Reid Simmons, Dimitrios Apostolopoulous, and William Whitaker. Autonomous navigation field results of a planetary analog robot in antarctica. In*Proceedings of the 1999 International Symposium on Artficial Intelligence, Robotics and Automation for Space*, Noordwijk,The Netherlands, June 1999.

[61] N. Muscettola. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, 1998.

[62] Nicola Muscettola. HSTS: Integrating planning and scheduling. In Monte Zweben and Mark Fox, editors, *Intelligent Scheduling*, pages 169–212. Morgan Kaufmann, San Francisco, CA, 1994.

[63] Karen Myers. Towards a framework for continuous planning and execution. In *Proceedings of the AAAI Fall Symposium on Disributed Continual Planning*, Orlando, FL, October 1998.

[64] Karen Myers. A procedural knowledge approach to task-level control. In *Proceedings of the Third International Conference of AI Planning Systems*, Edinburgh, Scotland, May 1999.

[65] I. Nesnas, M. Maimone, and H. Das. Rover Maneuvering for Autonomous Vision-Based Dexterous Manipulation. In *IEEE Conference on Robotics and Automation*, San Francisco, CA, April 2000.

[66] I. Nesnas and M. Stanišić. A robotic software developed using object-oriented design. In *DAC*, Minnesota, 1994.

[67] P. S. Schenker et al. FIDO Rover and Long-Range Autonomous Mars Science. In *Intelligent Robots and Computer Vision XVIII, SPIE Proceedings 3837*, Boston, September, 1999.

[68] S. Schneider et al. ControlShell: A Software Architecture for Complex Electromechanical Systems. *International Journal of Robotics Research*, 17(4), April 1998.

[69] M. Schoppers. A Software Architecture for Hard Real-Time Execution of Automatically Synthesized Plans or Control Laws. In *AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS)*, Houston TX, March 20-24 1994.

[70] R. Simmons. Structured Control for Autonomous Robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, 1994.

[71] R. Simmons and D. Apfelbaum. A Task Description Language for Robot Control. In *IEEE/RSJ Intelligent Robotics and Systems Conference*, Vancouver Canada, October 1998.

[72] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, J. O'Sullivan, , and M.M. Veloso. Xavier: Experience with a layered robot architecture. In *Proceedings of the First International Conference on Automoous Agents*, Marina del Rey, CA, February 1997.

[73] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Proceedings of the International Conference on Intelligent Robots and Systems*, Vancouver, Canada, October 1998.

[74] S. Singh et al. Recent Progress in Local and Global Traversability for Planetary Rovers. In *IEEE Conference on Robotics and Automation*, San Francisco, CA, April 2000.

[75] Steve Smith. OPIS: A methodology and architecture for reactive scheduling. In Monte Zweben and Mark Fox, editors, *Intelligent Scheduling*, pages 29–66. Morgan Kaufmann, San Francisco, CA, 1994.

[76] D. Stewart, R. Volpe, and P. Khosla. Design of Dynamically Reconfigurable Real-Time Software using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23(12), December 1997.

[77] E. Tunstel, R. Welch, and B. Wilcox. Embedded Control of a Miniature Science Rover for Planetary Exploration. In *7th International Symposium on Robotics with Applications, WAC'98*, Anchorage Alaska, May 1998.

[78] B. Underhill, A. Friedman, and et al. Three corner sat constellation: Management, systems, spacecraft bus, micropropulsion/payloads. In *Proceedings of the Thirteenth AUAA Conference on Small Satellites*, Utah, 1999.

[79] R. Volpe. Navigation Results from Desert Field Tests of the Rocky 7 Mars Rover Prototype. *International Journal of Robotics Research*, 18(7), 1999.

[80] R. Volpe and J. Balaram. Technology for Robotic Surface Inspection in Space. In *AIAA Conference on Intelligent Robots in Feild, Factory, Service, and Space (CIRFFSS)*, Houston, Texas, March 20-24 1994.

[81] R. Volpe et al. Rocky 7: A Next Generation Mars Rover Prototype. *Journal of Advanced Robotics*, 11(4):341–358, 1997.

[82] Rich Volpe, Sharon Laubach, Clark Olson, and Bob Balaram. Enhanced mars rover navigation techniques. In *Proceedings of the 2000 IEEE Conference on Robotics and Automation*, San Francisco, CA, April 2000.

[83] B. Wilcox et al. Robotic Vehicles for Planetary Exploration. In *IEEE Conference on Robotics and Automation*, pages 175–180, Nice France, May 12-14 1992.

[84] D. Woerner. X2000 Systems And Technologies For Missions To The Outer Planets. In *49th International Astronautical Congress/International Astronautica Federation*, Melbourne, Australia, September 28-October 2 1998.

[85] Y. Xiong and L. Matthies. Error Analysis of a Real-Time Stereo System. In *Computer Vision and Pattern Recognition*, pages 1087–1093, 1997.

[86] Y. Xiong and L. Matthies. Vision-guided Autonomous Stair Climbing. In *IEEE Conference on Robotics and Automation*, San Francisco, CA, April 2000.

[87] J. Yen and A. Jain. ROAMS: Rover Analysis Modeling and Simulation Software. In *nternational Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS'99)*, Noordwijk, Netherlands, June 1999.

[88] M. Zweben, B. Daun, E. Davis, and M. Deale. Scheduling and rescheduling with iterative repair. In Monte Zweben and Mark Fox, editors, *Intelligent Scheduling*, pages 241–256. Morgan Kaufmann, San Francisco, CA, 1994.