ARS ADVANCED ROBOTIC SYSTEMS INTERNATIONAL

Design Minimalism in Robotics Programming

Anthony Cowley[1]; Luiz Chaimowicz[2] & Camillo J. Taylor[1]
[1]GRASP Laboratory - University of Pennsylvania, Philadelphia, PA, USA.
[2]VeRLab - DCC - Federal University of Minas Gerais, Belo Horizonte, MG, Brazil.
{acowley, cjtaylor}@grasp.cis.upenn.edu, luizch@dcc.ufmg.br.

Abstract: *With the increasing use of general robotic platforms in different application scenarios, modularity and reusability have become key issues in effective robotics programming. In this paper, we present a minimalist approach for designing robot software, in which very simple modules, with well designed interfaces and very little redundancy can be connected through a strongly typed framework to specify and execute different robotics tasks.*
Keywords: *robotics programming, modularity, reusability*

1. Introduction

The evolution of robotics in terms of hardware and software design resembles the evolution of computer science in general. In the 60's, computer science was mostly focused on the hardware, and programs were developed for specific computer architectures, with little chance of reuse. In the last couple of decades, computer architectures, instruction sets and operating systems have become more stable and the focus has shifted from the hardware side to the software side. The development of efficient, modular and reusable code is now mandatory, and the use of modern software engineering is required in any system. This same process is happening now in robotics with the widening availability of stable robotic platforms.

At the GRASP Lab. – University of Pennsylvania, we have been developing ROCI – *Remote Objects Control Interface* (Chaimowicz et al. 2003; Cowley et al. 2004a, 2004b). ROCI is a software platform that provides numerous features to support distributed software design, and a programming model for creating reusable components called ROCI modules. Basically, a ROCI module encapsulates a process which acts on data available on the module's inputs and presents its results as outputs. They are self-contained and reusable, thus, complex tasks can be built by connecting inputs and outputs of specific modules.

One of the key design principles of ROCI is to keep the individual processing units (modules) simple, with well designed interfaces and no feature overlap. This minimalist approach allows modules to be easily tested, composed and reused in different scenarios. The main objectives of this paper are to discuss the genesis of this approach, its advantages in robotics programming and to describe its implementation in ROCI.

It is important to mention that in the last few years several different platforms have been proposed aiming to facilitate robotics programming. Some of these platforms, such as the *Player*/*Stage*/*Gazebo* framework (Gerkey, B., et al. 2001), are very general, allowing the simulation and execution of several applications using different robotic platforms. Others are more application specific such as JPL's *CLARAty* (Nesnas, N., et al. 2003) or CMU's *Carmen* (Montemerlo, M., et al. 2003) to mention a few. An overview of some of those platforms can be found in the eletronic proceedings of the workshop "Principles and Practice of Software Development in Robotics" (Brugali, D. & Reggiani, M., 2005).

We believe that the ideas presented in this paper can contribute to, and interoperate with, other platforms on the path to building more efficient and reusable robotic software. Specifically, ROCI provides an architecture built around a reflective type system that maintains pervasive object metadata. Using this foundation, ROCI components are capable of a level of automation, in the form of tools such as universal object serializers, formatters, and converters that require significantly less custom programming than they would with a less descriptive object model. This benefit is felt throughout component development because it leads to a minimization of repetition in code, and a distilling of component functionality free of the typical utility code necessary to fit the component into the platform.

2. ROCI Overview

ROCI is a self-describing, objected oriented, strongly typed programming framework that facilitates the development of robust applications for dynamic multi-robot teams. In ROCI, each robot is considered a node which contains several processing and sensing modules and may export different types of services and data to

other nodes. Each node runs a kernel that can be considered a high level OS. It mediates the interactions of the robots in a team, handling task allocation and execution, managing the network and maintaining an updated database of other nodes in the ROCI network.

A ROCI task is a way of describing an instance of a collection of ROCI modules to be run on a single node, and how they interact at runtime. It is defined in an XML file which specifies the modules that are needed to achieve the goal, any necessary module-specific parameters, and the connections between these modules. At runtime, the kernel downloads any components not stored locally, loads the necessary libraries into memory in a separate system process, and creates the inter-module connections through a Pin architecture that provides a strongly typed, network transparent communication framework. No extra compilation step is necessary to link several objects, which is common in frameworks that deal with several source code libraries. A new task in ROCI can be "built on-the-fly", dynamically connecting independent modules on a single node, and over a network. A good analogy is to view each of these modules as an integrated circuit (IC), which has inputs and outputs and does some processing. Complex circuits can be built by wiring several ICs together, and individual ICs can be reused in different circuits. Fig. 1 shows a diagram of this architecture.

The main interface between a human operator and the robot team is the ROCI Browser. The browser displays the multi-robot network hierarchically: the operator can browse nodes on the network, tasks running on each node, the modules that make up each task, and Pins within those modules. The browser's main job is to give a user command and control over the network as well the ability to retrieve and visualize information from any one of the distributed nodes. Specifically, using the browser, the user can start, stop and monitor the execution of tasks in the robots remotely, change task parameters, send relevant control information for the robots and display the outputs of Pins for which visualization routines exist. Also, elaborate missions can be constructed within the browser using scripts. Mission scripts can be generated online or offline, and specify a sequence of actions that should be performed by a team member.

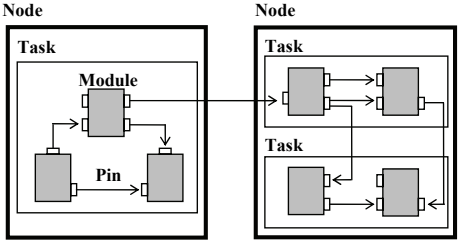

Fig. 1. ROCI Architecture: tasks are composed of modules and run inside nodes. Communication through Pins can be seamlessly done between modules within the same task, modules in different tasks, or in different nodes.

ROCI is still under development and one of its current limitations is the lack of support for a real time OS. Nevertheless, as shown in this paper, ROCI has been successfully used for programming different robots in several applications.

3. Modularity and Reusability

3.1. Component Design

The reuse of components has been a central issue in software design for many years. In the interest of furthering the cause of modular, plug-and-play style software reuse, many paradigms, methodologies, toolkits and best practices have been proposed, with each successive generation of engineers building on the work of their predecessors. Therefore, the ideas that we present here should at once feel both familiar and refreshing to the experienced computer user frustrated with the state of software engineering in the field of robotics.

As described earlier, the building blocks of the ROCI system are ROCI modules. These elements, written by users, create the language of the ROCI network. That is, we do not view a node on the network as being programmed in a system language, but rather in a high-level, domain-specific language defined by a collection of modules. This language is to be specified and built by the users themselves through the creation of a working set of modules. Without user-generated content, the entire burden of language primitive creation lies with the platform designers, which would either limit the effective domain of the language to the areas where the architects chose to focus, or inflate algorithm expression by relying on a generic vocabulary. However, extensibility, especially when treated as an essential element of the system, places a significant burden on the extension author. In the ROCI setting, where all functionality is pushed into the modules, née extensions, it is essential that strong conventions and guidelines be provided to aid the user in creating interoperable, reusable components.

3.2. Designing to an Interface Specification

A popular method for managing development teams is to jointly, or dictatorially, establish the interfaces through which each component of the system should interact. With such a specification in hand, responsibility for developing the individual components of the system can be distributed among the team with some hope that when everyone is done, all the parts will work well together. Eliding the difficulties of system integration for the time being, there is promise in this managerial tactic as it establishes boundaries for each component; boundaries that not only help delineate functionality, but also provide natural latches to use for building tests. Also, the process of interface design offers an opportunity for system architects to reason about the system at a very high level. During this stage of development, where large

pieces of the system exist as nothing more than elements in a block diagram, the entire system can be taken in at once, and re-organized if needed.

The danger with designing to an interface is that what happens behind the interface, which is ostensibly of no interest to potential customers of the interface, can lead to situations where the internal system only meets the requirements of the interface by coincidence. That is, the component may not correctly capture, model, or reflect the desired logic, but happen to produce the desired output where it intersects with a limited interface. This situation may occur in a component that functions absolutely correctly for its originally specified range of inputs, so it will not be caught in testing.

A lack of fidelity in the manifestation of an abstract concept in the form of software may not matter if the specified interfaces are completely verified over the entire possible input range. However, should a component have a large number of interfaces, the decision to allow one more is often an easy one to make. This new window into the logic encapsulated by the component may allow some part of the hidden, and heretofore irrelevant, shortcomings of the software implementation to spill out. Alternatively, a component with multiple interfaces may become a target for refactoring into smaller sub-components. Here again, the split may reveal implementation problems that weren't visible in the originally tested component. The problem is not one of engineering, and can fairly be restated as trying to use something in a way that it was not designed for. If a component was designed to evaluate a particular function at two different points, asking it to evaluate the function at an unconsidered third point can be seen as unfair by the engineer who implemented the component or entirely reasonable by an architect who asked for an implementation of a particular function. The breakdown occurs when the consumer of the component, in this case the architect seeking to extend the system's functionality, does not respect the interface contract adhered to by the implementer because that limited interface is not related to the canonical concept the component was intended to capture. At its root, the problem is due to inconsistencies between the architect's ontology and the engineer's reification pragmatism. Note that the two parties may find themselves with different priorities: the architect is more productive when she can work with a clean domain ontology, while the engineer can make the component function most efficiently when it meets a detailed set of specifications exactly. The development of specifications then bears the burden of responsibility for making it possible for the architect to work at a high level while still keeping component design as efficient as possible.

Clearly, trouble may be avoided if component refactoring is kept to a minimum. What is needed is a working set whose each component is generally applicable, yet individually inviolate. While freezing component interface design is a straightforward policy to implement, it leads to functional overlap, and the associated interface divergence, if components are found to be inadequate in and of themselves as well as in combination. Thus, the language used to specify the system must be built upon primitives that fully cover their stated operating domain. One logical response to this position is to design a reduced instruction set, with a mind for great efficiency at the instruction, i.e. component, level and functional efficacy achieved through complex component composition. This is arguably the safest strategy to take, but it represents a complete departure from the drive to define a high-level domain language. Instead, we have settled on a primitive design that is powerful enough to clearly and succinctly express complex, decentralized robot behaviors while still retaining the qualities of efficient, irreducible components.

3.3. ROCI Module Design

The basic granularity of the language we have chosen to consider is found in the transformation of one type of data into another. Components that follow this pattern typically have the favorable quality of being context-free: they can fully describe their own functionality, and are idempotent with regard to usage scenario. Put simply, a data translator always translates a specific input data type to a specific output type, no matter where the input comes from or where the output goes to. This has the further benefit of letting us reason about component usage solely by the intrinsic properties of input and output types. Properties of the high-level system itself are expressed through the chaining together of multiple translator components. Finally, and most importantly, the prototypical translator component only needs to know about its own state, two types of data and those algorithms related to translating between those two types.

Small scale component design is an important part of the attitude we wish to foster in our developers. We want developers to focus on creating small, stand-alone processing loops that do one thing and do it well. This is a similar philosophy to the traditional UNIX shell design, where small, focused utilities are glued together in an ad hoc fashion through the use of pipes. The benefits of such a design are many, but primarily we wish to avoid feature overlap and component complexity.

Feature overlap occurs when multiple components are capable of doing the same thing. The danger here is not only confusion when overlapping components appear in the same project, but also a duplication of development effort and a greater chance for the aforementioned interface divergence. Redundant component functionality is an obvious inefficiency, but interface divergence is

often the more dangerous of the two problems. This phenomenon occurs as basic functionality is expanded upon in different places in parallel. The end result is that there exist multiple ways of doing very similar things, but the different methods are not entirely compatible. This leads to correct usage patterns being applied incorrectly due to a change in which component provides the service; a change which can go unnoticed due to the overall similarity in functionality.

Component complexity rises when a developer continues adding functionality to a single module in order to supposedly accomplish a near-term goal more rapidly. The result is that the component often becomes brittle and difficult to test. The brittleness comes from unchecked intra-component dependency growth, wherein each part of the component is dependent upon one or more other parts; a problem typical of monolithic designs. Testing difficulty is related to the number of unit tests required to adequately confirm a component's correctness, a number that rises combinatorially as functionality is added. The watchwords here are, predictably, "keep it simple" (Berners-Lee, 1998). Nobody wants to use a component that includes unnecessary features that may adversely affect the stability of the system under construction.

3.4. Interface Specification as Type Definition

The difficulty with an essentially untyped interface, as with UNIX shell pipes, is that the system architect has less to work with when figuring out how to put components together than the design-by-interface scenario we have considered up to this point. For this reason, and the way interface specification can come to dominate the description of translator component functionality, we make extensive use of the strong static type system of the .NET CLR. This platform lets us inspect a compiled module to ascertain all necessary information about input and output types to be able to design at the system level relying primarily on type matching. A typical program, or task in the parlance of ROCI, is designed by connecting the interfaces of existing modules and hypothetical modules such that sensor-derived, raw data is transformed into forms that can be consumed by high-level modules that implement a particular algorithm, and then transformed back into hardware platform-specific forms. The important part of this development process is that the hypothetical modules have their interfaces, defined by the required data type translation, specified once the system schematic is drawn. The module now has its boundaries, and can be developed and tested in isolation from the system it will ultimately find itself a part of. Most importantly, the definition of the relevant data structures should greatly inform the design of the module that uses them. We believe that the structure of the functional part of a component should mirror the structure of the data model. While we still have to take care in defining interfaces that satisfy both system architects and component engineers, by focusing on the notion of type translation we can isolate the component from any context it may appear in. Furthermore, the difficulty of exhaustively covering the input domain, so as to avoid future refactoring, is simplified through the use of object oriented programming techniques such as interface inheritance.

3.5. ROCI Pins

The statically typed interfaces that glue ROCI modules together are ROCI Pins. Creating a Pin involves declaring a data structure separate from any module; all functionality is provided by base classes. Making these data structures first class entities in the system serves a number of purposes. First, the Pin base class provides a wealth of functionality that encourages their adoption; features such as network transparency, connection optimization, data buffering, throughput monitoring, subscription-based data flow, and automatic, network-safe, polymorphism make Pins a powerful interface mechanism. Second, declaring these interfaces as separate, re-usable components encourages their re-use. This not only saves development effort, but also provides a common interface toolbox for component authors to work from so that their modules may be used in concert. Third, the need to explicitly declare the structure of an interface forces the component designer to consider the data model his component operates upon.

The data structures embodied by ROCI Pins can represent the native data structure that an algorithm operates over, the data generated by a sensor, or the data understood by a hardware device. Time spent on clarifying this data structure is time well spent, as it should inform the structure of the functional parts of the modules that produce or consume the data. In addition to encouraging functional structure to mirror data persistence structure, the explicit construction of Pins discourages the unchecked interface addition common to standard class design. When designing a class to be consumed locally, or remotely via remote procedure call / remote method invocation, developers tend to add public interfaces in an ad hoc fashion as the need arises. The organization of these interfaces is not related to any underlying data model, but is instead mediated by a particular usage scenario. Since we wish to develop reusable components, the most sensible interface organization is one that is patterned on the component's data model, a property intrinsic to the component and independent of any usage scenario.

3.6. The ROCI Development Process

Developing a ROCI task involves determining the desired behavior, and constructing a block diagram of the necessary components. This will include modules that generate data, either by interfacing with sensors or

through some procedural method, modules that implement an abstract behavioral algorithm or controller, and modules whose output passes back into hardware or a simulation layer. At this point, the interfaces between these modules may well not match up, and the system architect will need to specify the additional translations necessary to match input types to output types.

Each module in this graph should represent a one-to-one translation of input to output, or an interface composition of multiple simple types into a more complex type. Importantly, multiple inputs or outputs should always represent a data structure composition or decomposition, respectively, and not the presence of varied, disparate functionality encapsulated in a single Module. The specification of this graph, much like a circuit diagram, will usually involve the reuse of many existing modules. In fact, one can evaluate this initial design by ensuring that module reuse increases as one reads the design from high level to low. That is, a specific robot behavior may need to be written for the task under construction, but most of the mid-level translators should be existing modules, and nearly all the hardware-interface modules should be pre-written for a stable hardware platform.

We refer to the job of ROCI task specification as architectural, and the job of implementing ROCI modules as engineering. The distinction is not meant as one of personnel (indeed the typical roboticist will act as both system architect and low-level engineer), but is instead used to delineate the level of programming being applied. ROCI tasks are compositions of modules, and are written in a high-level declarative language. The modules themselves are written in system languages such as C/C++, C#, Java, VB, etc. We especially encourage team cooperation and discussion during the architectural design phase of system development. The notions of modules and Pins provide a guideline for this discussion, and offer a natural point of departure for the team to split up and start writing code.

4. Example

The ROCI task that implements the robot behavior of following a colored blob provides a good demonstration of ROCI design. This task has been successfully executed with different robots as part of the DARPA MARS2020 project (Chaimowicz et al. 2005) Each of the following sections briefly describes a component module used in the task and how it has been constructed with generality in mind. A diagram of the module connections is shown in Fig. 2. It is important to note that some of these modules have also been used in other tasks, which demonstrates the benefits of the component design. For example, the "Blob Extractor" was used for identifying targets in a surveillance task and the "Stereo Cam" as a sensor for obstacle avoidance.

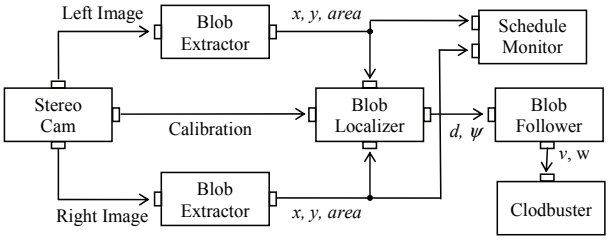

Fig. 2. Diagram of a ROCI task that implements a blob following behavior. The modular design allows individual components to be easily reused in other tasks.

4.1. Stereo Camera Driver

A module was written that interfaces with the system-level APIs for a stereo color camera. This module reads the camera's intrinsic parameters from a proprietary API, and captures images once the camera has started. The module is controlled through a number of startup and runtime parameters that govern properties such as color processing, capture resolution, and capture rate. The primary function of the module is to capture images and export them in a general video frame data structure that captures information related to image capture in a format independent from the particular camera being used. This module is used in any situation where the particular camera is installed, and its output can be processed by any module that consumes the video frame interface.

4.2. Blob Extractor

A mid-level data translator related to this task is the module class that extracts color blobs from video frames, and exports 2D blob information. This module translates raw video frames into blob content information, and is used in any situation where blob extraction is useful, independent of the hardware platform, be it a UGV, UAV, or fixed camera installation. The module works by comparing the image pixels with a pre-computed lookup table in order to segment the regions that match the specific colors and compute the blob information.

4.3. Stereo Blob Localizer

This mid-level data translator was written specifically for this task. It consumes the outputs from a pair of blob extractors and generates a 3D localization of each blob in a camera coordinate frame. The one-to-one translation performed by this module is from a data structure containing a pair of blob content information structures to a single data structure that contains 3D blob localization information. In this case, the composition of two 2D blob structures into a single structure is collapsed into the module that also generates the 3D record. This collapse is allowed because the structure composition is trivial and the resultant data type, a pair of epipolar 2D blob structures, is somewhat unusual. Importantly, the presence of multiple inputs does not imply that the stereo blob localizer could be refactored to split its functional structure.

4.4. Blob Follower

Another module written specifically for this task, this component implements the original desired behavior. We implemented a distributed leader-follower controller that tries to keep the follower within a desired distance (both in x and y) from the leader, as shown in Fig. 3. The controller generates linear and angular velocities for the follower based on its distance and bearing to the leader. Thus, this module can accurately be viewed as converting stereo blob localizations into robot motion commands. At this point in the design hierarchy, both input and output types are completely independent of target platform or usage scenario, thus leaving the component's functionality completely portable.

Fig. 3. Two Clodbusters in a Leader-Follower formation. The robot on the left is tele-operated while the robot on the right is executing the blob follower behavior.

4.5. Clodbuster

A low-level module that is involved in one version of this task contains physical characteristics and mappings for a particular experimental UGV platform known as the Clodbuster. This module is responsible for converting motion commands given in real-world units into the unit-less servo commands that actuate the robotic platform. Another version of this task replaces this module with one that interfaces with a Segway RMP. This module substitution is the only change necessary to retarget the task between the two platforms. Since the task itself is built in a high-level domain language whose elements are the modules listed here, no system level programming needs to be done to change hardware platforms. Instead, the high-level program needs to have one declaration type changed.

4.6. Schedule Monitor

As a demonstration of ROCI's reactive scheduling capabilities, a custom performance monitor was added to this task long after its original design. This monitor consumed the output of the blob extractors to determine when a target was spotted. Upon target identification, the schedule monitor would instruct ROCI to apply a task schedule that prioritized the vision-related modules over other sensing modules. The addition of this, a high-level, very context-specific module required no system-level programming outside this module's development; all the existing pieces could be reused without change.

5. Conclusion

This paper presented a minimalist approach for designing software in robotics. The key idea can be summarized by the watchwords "keep it simple". Basically, we develop modules with well designed interfaces that perform very specific functions, with no feature overlap with other modules. Module developers are free to concentrate solely on the new functionality offered by their modules as basic interconnection functionality – i.e. Object serialization, formatting, and type coercion – is handled by system-level type-aware translation components that can process arbitrary data types. These modules are connected in a high-level fashion through a strongly typed framework, providing the desired functionality for each scenario. We have been applying this design paradigm in the development of ROCI, a programming framework that has been used in the implementation of a variety of robotic applications. We believe that this type of approach is strongly recommended in order to have modular, reusable code that can be efficiently developed and easily tested.

6. References

Berners-Lee, T., (1998). Principles of Design. http://www.w3.org/DesignIssues/Principles.html.

Brugali, D. & Reggiani, M., (2005). Principles and Practice of Software Development in Robotics. ICRA2005 Workshop. http://robotics.unibg.it/icra05ws/

Chaimowicz, L., Cowley, A., Sabella, V., and Taylor, C., (2003). ROCI: A distributed framework for multi-robot perception and Control. In: *Proc. of the 2003 IEEE/RJS IROS*, pp 266-271.

Chaimowicz, L., Cowley, A., Gomez-Ibanez, D., Grocholsky, B., Hsieh, M., Hsu, H., Keller, J., Kumar, V., Swaminathan, R., and Taylor, C. (2005) Deploying Air-Ground Multi-Robot Teams in Urban Environments In: *Proc. of the 2005 International Workshop on Multi-Robot Systems.*

Cowley, A., Hsu, H., and Taylor, C., (2004a). Distributed sensor databases for multi-robot teams. In: *Proc. of the 2004 IEEE ICRA.*

Cowley, A., Hsu, H., and Taylor, C., (2004b). Modular programming techniques for distributed computing tasks. In: *Proc. of the 2004 Performance Metrics for Intelligent Systems (PerMIS) Workshop.*

Gerkey, B., Vaughan, R., Støy, K., Howard, A., Sukhatme, G., and Mataric, M. (2001). Most Valuable Player: A Robot Device Server for Distributed Control. In: *Proc. of the 2001 IEEE/RSJ IROS*, pp. 1226-1231.

Montemerlo, M., Roy, N., and Thrun, S. (2003). Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit''. In: *Proc. of the 2003 IEEE/RSJ IROS*. pp 2436-2441.

Nesnas, N., Wright, M., Simmons, R., Estlin, T., and Kim, W. (2003). CLARAty: An Architecture for Reusable Robotic Software," *Proc. of SPIE Aerosense Conference.*