ROCI: Strongly Typed Component Interfaces for Multi-robot Teams Programming

Anthony Cowley[1], Luiz Chaimowicz[2], and Camillo J. Taylor[1]

[1] GRASP Laboratory – University of Pennsylvania, PA, USA
{acowley,cjtaylor}@grasp.cis.upenn.edu
[2] Computer Science Department - Federal University of Minas Gerais, MG, Brazil
chaimo@dcc.ufmg.br

1 Introduction

The software development process in robotics has been changing in recent years. Instead of developing monolithic programs for specific robots, programmers are using software components to build all kinds of robotic applications. As discussed in Chapter *Trends in Component-Based Robotics* of this book [Bru06], component based development offers several advantages such as reuse of code, increased robustness, modularity and maintainability.

Considering this basic guideline, we have been developing ROCI – *Remote Objects Control Interface* [CCS2003, CHT2004b]. ROCI is a software platform for programming, tasking and monitoring multi-robot teams. In ROCI, applications are built in a bottom-up fashion from basic components called ROCI modules. A module encapsulates a process which acts on data available on its inputs and presents its results as outputs. Modules are self-contained and reusable, thus, complex tasks can be built by connecting the inputs and outputs of specific modules. We can say that these modules create the language of the ROCI network, allowing task designers to abstract from low level details and focus on high level application semantics. ROCI is specially suited for programming and monitoring distributed ensembles of robots and sensors, since modules can be connected across local networks in a transparent way and data can be transmitted and visualized in different formats.

One key characteristic of ROCI is its approach for developing robust interfaces to connect individual modules. In component based development, external interfaces should be clearly defined to allow an incremental and error resistant construction of complex applications from simpler, self-contained components. By making interfaces explicit and relying on strongly-typed, self-describing data structures, ROCI allows the development of robust applications. Moreover, ROCI's modularity supports the creation of parallel data flows which favors the development of efficient applications.

D. Brugali (Ed.): Software Eng. for Experimental Robotics. STAR 30, pp. 169–182, 2007.

In this chapter we present an overview of the ROCI framework and discuss in more detail its modular, strongly-typed design. The chapter is organized as follows: the next section describes the ROCI framework, giving an overview of its architecture and main features. Section 3 describes in more details some of ROCI's benefits for programming distributed robots. Specifically, we discuss modularity, strongly typed interfaces, parallelization, logging and web accessibility. In Section 4, we present a couple of examples that illustrate some these benefits and, in Section 5, we conclude the paper discussing some of ROCI's main contributions.

2 ROCI Overview

ROCI is a dynamic, self-describing, object-oriented, strongly typed programming framework for distributed robot teams. It provides programmers with a network transparent framework of strongly typed modules - assemblies of metadata, byte code, and machine code that can consume, process and produce information. ROCI modules are injectable (they can be automatically downloaded and started on a remote machine), reusable, browseable, and support automatic configuration via XML. These features, coupled with a dynamic database of available nodes and network services, allows a programmer to write code that utilizes networks of robots as resources instead of independent machines.

2.1 Modules and Tasks

The building blocks of a ROCI application are ROCI modules. Each module encapsulates a process which acts on data available on its inputs and presents its results on well defined outputs. In ROCI, there is no specification in the code relating to where input should come from or where output should go; the only specification is the type of data each computational block deals with. The key design principle of ROCI is to keep 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 [CCT2006].

Complex tasks can be built by connecting the inputs and outputs of specific modules. 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. A good analogy is to view each of these modules as an integrated circuit (IC), that 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.

As will be explained in the next section these connections are made through a pin architecture that provides a strongly typed, network transparent communication framework. Pins can connect modules within the same task on

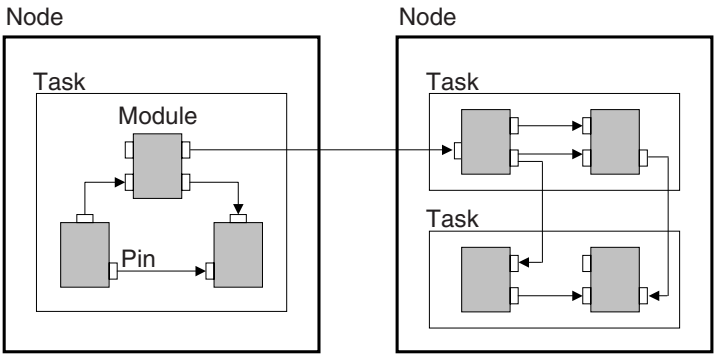

Fig. 1. ROCI Architecture: tasks are composed of modules and run inside nodes. Communication through pins can be seamlessly done between modules within the same task, modules in different tasks, or in different nodes.

the same computer, between tasks on the same computer, or between tasks on different computers. Figure 1 shows an example of this organization.

2.2 Pin Based Communication

The wiring that connects ROCI modules is the pin communications architecture. Basically, a pin provides the developer with an abstract communications endpoint. Pins in the system are nothing more than strongly typed fields of a module class, and thus connecting a producer pin to a consumer pin is as simple as setting a reference to the producer in the consumer's field. One can set a pin's input in two distinct ways that are each useful in different situations. On a lower level, pins can be connected dynamically during program execution. This can be accomplished by querying nodes on the ROCI network for available pins – usually with a type constraint – and may involve dynamically creating local pins to bind to the discovered remote pins. However, the simpler way of binding statically declared pins together is via the XML descriptions that define ROCI tasks.

As the producer generates data it assigns an updated reference to the latest data to its pin. This assignment causes the pin to fire messages to all of the registered consumers in the network, alerting them to the availability of fresh data. A typical usage of this system is for a module to make a blocking call to one of its input pins which returns once the input pin has gotten new data from the output pin it is registered to. Alternatively, a module can ask its input pin to copy over whatever data is available immediately, whether it is new or not. Pin data is time stamped, allowing the consumers to determine how current their data is. Together with logging mechanisms, the time stamps also allow the creation of sensor databases and the execution of distributed queries [CHT2004].

It is important to stress the advantages of such strongly typed communication channels. In ROCI, the data exchanged between modules are highly structured entities and not simply byte streams. This ensures that valid connections are made between different modules, increasing robustness. Also, it allows the development of specific "translator" modules based on these types that, for example, automatically format the information for an human operator. Additionally, the system supports what we call *blind polymorphism* in which the data being exchanged inherits from some base class and the upcast is made transparently on the source node, not on the destination. In this way, a producer module that exports a very complex data type that inherits from a more basic one will transmit an instance of the relevant, smaller, ancestor class to any consumer module that only needs the basic information. The blind polymorphism guarantees that only the necessary data is transmitted, and that the receiver need not be updated to parse new data types that it does not directly make use of.

2.3 ROCI Kernel

The kernel is the core control element of ROCI. It can be considered a high level OS that mediates the interactions of robots in a team. The kernel manages the Remote Procedure Call (RPC) system, the real-time network database, module and task allocation and injection, and a RESTful web service interface for remote monitoring and control. There is a copy of the kernel running on every entity that is part of the ROCI network (robots, remote sensors, etc.).

The system management functionality provided by the kernel focuses on resource discovery, usage, and prioritization. Peer-to-peer file discovery and sharing are included to facilitate automatic distribution of injected tasks. Usage statistics are monitored, and controllable, for both CPU and network load. These loads are monitored at the scale of modules and pins, respectively, thus allowing for an appropriately fine grain of control over resource allocation that is completely dynamic and modifiable at runtime.

For example, when the user requests a task be started on a given node, the kernel running locally on that node first ascertains whether or not the proper versions of component modules are available locally. If they are not, it queries the network for a node that does have the modules in question and downloads them automatically. Once the byte code of the modules that comprise the task all exists locally, the task is loaded. Resource allocation priorities may be set at startup, but they may also be modified at runtime by any module, thus allowing for reactive, context-aware, resource allocation. An example of such a dynamic behavior is one in which CPU resources are largely devoted to obstacle avoidance while a robot moves quickly, but shifted to target detection as the robot's rate of motion decreases.

The ability to support this type of flexibility is characteristic of ROCI's data-driven design. On the far end of the flexibility-spectrum, all the data

that flows through ROCI is accessible as XML through a web browser. Towards the efficiency end of things, this same data is internally shared through more compact, event-based binary serialization. The pervasive use of metadata throughout the system allows for this style of multiple views of the same data. There is very little effort required to keep, for example, binary and XML serialization code paths in sync because translation between these forms, as well as HTML, Email, IM, SMS, etc., occurs automatically based on the metadata. The ultimate win is that the minimal metadata necessary to effect basic translation and formatting of data structures is present in the type declarations themselves. The reflective type system means that there is never an unidentifiable byte stream; instead, the programmer effort put into creating data structures is exploited again on the front-end to structure and label live data.

2.4 ROCI Browser

The job of presenting this network of functionality to the user falls upon the ROCI Browser. The browser's main job is to give a user command and control over the network as well as the ability to retrieve and visualize information from any one of the distributed nodes, and make informed decisions about network operations. 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.

Using the browser, the user can start, stop and monitor the execution of tasks on the robots remotely, change task parameters, send relevant control information for the robots or even to tap into and display the outputs of pins for which display routines exist. Since modules and pins are self-describing entities, when the user browses through the tasks, he or she can immediately have a complete description of the modules and pins in use. Given this information, the browser can automatically start appropriate modules locally to tap into the remote data for visualization or processing purposes. This can be very useful for debugging purposes during development and for situational awareness during deployed execution. Figure 2 shows a snapshot of the browser during the execution of a multi-robot mission. In this mission, a team of robots should search an urban environment for a specific target while maintaining network connectivity and send a picture back to a base station [CCG2005].

Elaborate missions may also be constructed within the browser using *scripts*. ROCIScripts echo the domain language foundations of the ROCI task development process. In task development, semantically meaningful constructs such as modules and pins are reasoned about. Similarly, ROCIScript is based on the notion that the language in which a human commands a robot should be a conversation at an appropriately high level of abstraction. Furthermore, the specific abstractions relevant to a particular application may be used to define a compact working vocabulary. Rather that pre-determine a fixed-syntax

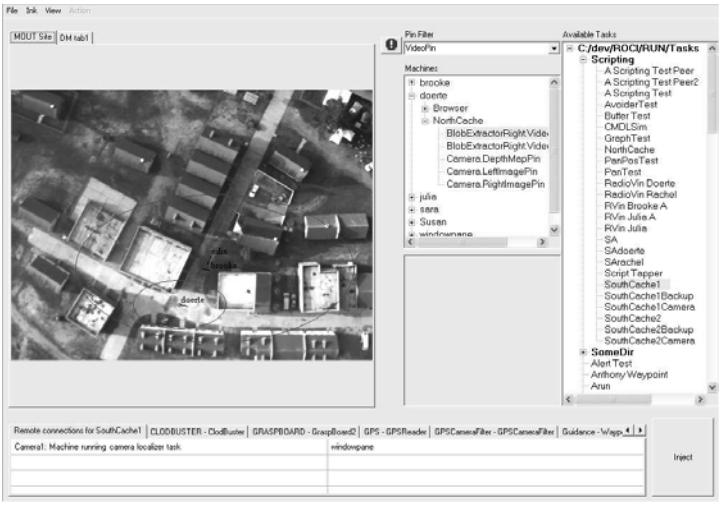

Fig. 2. ROCI Browser during the execution of a multi-robot mission: It is possible to monitor the status of each node, the tasks and modules being executed and even see the real time location of each robot on a map.

scripting language for robotics, ROCI extracts the scripting commands a robot supports from the modules that make up a task. In this way, the vocabulary used to script a particular task is always relevant to that task: the user is given a meaningful set of commands to work with for a given robot, and the robot runtime only need handle commands that are actually relevant to its capabilities. The scripting system lets developers create high level behavioral scripts that may be re-parameterized by any software package capable of editing XML. This parameter assignment may be human operator-driven through the browser, or it may be accomplished by another piece of software that can then inject the new script into the robot. The script interface thus represents a very high level of robot interfacing: commands such as *Search Room* or *Go To Waypoint* are implemented as scripts that may be invoked from a wide variety of programming languages, environments and GUIs.

3 Programming the ROCI Platform

The components of the ROCI framework presented so far make up a system that encourages a modular design style based on the construction of loosely coupled components. While the benefits of such design techniques are manifold, we have found that the interaction between software and well-designed robot hardware provides a fresh perspective that clarifies the benefits of this technique while simultaneously making the path to implementation more obvious.

3.1 Modular Design

ROCI modules, written by users, create the language of the ROCI network. This short statement touches upon two key issues: users are better positioned to establish their working language than a central ROCI development authority, and the domain language established can be far more powerful than any fixed system language. The intuition of the latter statement motivates the former. The desire to have non-kernel developers determine the primitives of the framework demands that the system be built around structures and standards that aid this effort. To this end, ROCI defines a module development model where the boundaries of an object, its interfaces, place bounds on its functionality. In other words, no module should contain functionality that could not be generally inferred from an inspection of the module's pins. Furthermore, we encourage developers to minimize the number of moving parts within a single module. This method of focusing the utility of any given module actually makes it far easier to describe what a module does to other developers, encourages reuse by not including unwanted functionality, minimizes functional overlap with other modules, eases task refactoring, and makes code maintenance simpler.

As discussed in [CCT2006], the watchwords in ROCI module design are "Keep It Simple". Developers should focus on creating small, stand-alone processing loops that do one thing and do it well. 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 something we refer to as 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.

3.2 Strongly Typed Interfaces

Abstractly, one can view a ROCI task as being built on top of two distinct levels of protocol design. There is a high level application protocol defined by how the various modules are connected and ordered in the parallel data flows described above. This protocol is created by the task designer in simple declarative fashion in the task XML file. The low level protocol is established by pin type declarations that define how connected modules communicate, but say nothing for how end-to-end data flow is accomplished. Critically, the details of the low-level protocol are transparent from the high-level, and the high-level data flow protocols are invisible from the module level. This separation allows for the easy creation of application-level protocols, i.e. data flow patterns, which can be distributed to take advantage of whatever parallelism is offered by the unique IO patterns and priorities of the given application.

Viewed another way, semantic significance is created at the application protocol level. It is at this level that developers connect, for example, Position Pins or Video Pins. These constructs define a working set of ideas that are meaningful in the specific context of the application, as opposed to any sort of pre-defined language. Furthermore, they may properly be reasoned about at the application level. That is, the task developer need not be concerned with bits on the wire. Instead, the task designer works only with high-level, complex data structures and terminology specific to the desired behavior. ROCI pins, on the other hand, are concerned with establishing the bit-level protocol; how data is passed between functional elements. This protocol is derived from the structure of the pin data itself. The programmer defines various structures that allow for a clean, single-argument method invocation syntax, and this data structure definition work is exploited to define wire protocols. ROCI determines if there is any type ambiguity involved in a transaction between two connected pins, and, if so, is able to insert the necessary type metadata into the byte stream. For example, value types in the .NET framework are transferred by direct binary serialization while instances of reference types that are used polymorphically at runtime may be serialized, but must also be tagged with the metadata necessary for the receiver to parse the incoming data. This transaction protocol is automatically created by ROCI through type reflection.

3.3 Parallel Computing

Modular design promotes a style of application architecture that scales well, and is a natural fit for distributed computing. Whether the distribution is across a network or across a multi-core CPU, breaking an application into discrete, potentially-parallelizable parts is an important technique given modern hardware. Currently, many parallelization efforts focus on a particular application and a single target platform rather than more general techniques. Such ad-hoc efforts typically involve breaking a sequential application design

into two or three minimally synchronized flows. Instead of focusing on such a top-down approach, we advocate a bottom-up approach that does not necessarily consider any particular hardware or application. Instead, we consider the potentially-parallelizable parts in isolation with the focus on functional integrity rather than parallelization.

The difficulty in actually determining the potential threading granularity of an application is, unfortunately, considerable. Indeed, significant portions of many applications have such rigid data dependencies that there is no way to avoid inherent linearity in processing. The field of robotics, however, offers new insight into the question of when individual application building blocks may execute in parallel via its characteristic utilization of multi-channel, variable rate IO.

Data input is, intrinsically, a blocking operation. It is an opening in the loop of a program that the calling thread may not have control over. Thus it makes sense that when one thread can not proceed for want of input, another thread that is not starved of data should proceed. The alternative of polling each sensor once per loop iteration limits all processing to the rate of the slowest sensor. Furthermore, such an architecture imposes constant latency across all data processing activities. Instead, we can easily address the natural desire to better utilize processing resources by addressing the flow that has data ready to be processed. Once we have broken the application into multiple parallel streams, we can also control, to a certain extent, the rate of iteration for each flow to better address latency requirements of the system. For example, Figure 3 shows a diagram of a typical robotic application: a robot has to maintain a particular heading until a color target is spotted. In this case, we can take advantage of multiple processing units and conform to the naturally asynchronous pattern created by the input devices: a high-frequency magnetometer and a low-frequency imaging sensor. Importantly, the only module that needs to be at all thread-aware is the motion planner that implicitly constructs an application-specific compound data type out of bearing and image data. As long as this fusion module maintains some imaging state, the motors can be controlled at the rate of the higher-frequency sensor.

Parallel data flows are characteristic of ROCI tasks. Each task can be seen as a collection of data flows that always progress forward, sometimes joining to form new streams. The joining points of these data flows define an application-specific protocol that, ultimately, describes what the application does but is entirely absent at the module level. In this fashion, we have effected a clean separation between application-level architecture and building block construction.

3.4 Logging

The same idea behind the creation of application-protocols through the composition of low-level type-based protocols can be seen in the way data logs are implemented in ROCI. Rather than jumping straight into a compound

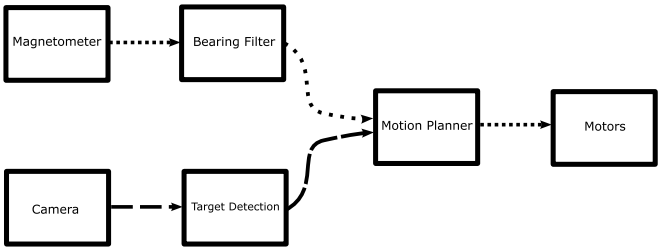

Fig. 3. Parallel data streams present a natural separation scheme for multi-threading. The dotted lines in this figure denote high-frequency data transactions, while the dashed lines represent relatively slow data flow.

logged data type, each data flow remains simple until a more complex type is required. By efficiently cross-indexing multiple simple data logs, we can provide compound data types that do not require a fixed schema, or any changes to the providers of the low level data. The compound data type returned is defined by the query itself, thus cleanly separating such high-level data mining from the low-level code that generates the data.

This system is again built around the reflective type system. A universal logging mechanism can tap into any pin connection, ascertain what type of data is being transmitted, and log all the data with type-appropriate indexing. These loggers can then describe the data that has been logged so that complex queries can be constructed that find data stores on the ROCI network and combine them in novel ways. Such a logging mechanism is completely reliant on all data being self-describing. Without this metadata, loggers for each data type would need to be written. The smart, type-aware logging process lets the development team add features to all type logging at once while also making the code maintenance job tractable.

3.5 Web Accessibility

As mentioned earlier, ROCI's data-driven internals are completely accessible to network peers. The deep extent of data accessibility is a manifestation of the desire to integrate ROCI with the broadest possible spectrum of other software packages as well as hardware devices. To this end, ROCI supports the standards and protocols of the Internet: HTTP and XML. A human operator may navigate a ROCI node with a web browser as the data content is converted from raw XML to formatted HTML through the use of XSLT and CSS. Machine peers, meanwhile, may consume the raw XML documents using standard HTTP methods.

This data accessibility is entirely provided by ROCI itself, and not dependent on developer support. While individual formatting methods may be overridden, developers can benefit from various Pin Exporters capable of operating on any ROCI pin. This creates an environment of universal accessibility, and allows the development team to support new interoperation technologies without having to update every already-defined interface. Instead, new translators may be developed that bring all ROCI data to a new format or protocol.

4 Examples

4.1 Module Reuse: Indoor/Outdoor Navigation

As discussed in this chapter, ROCI's modular design and strongly typed interfaces make component reuse a simple job. Here, we present an example in which a trajectory controller module was used to navigate different robots in different scenarios. Basically, this module receives two inputs: the robot pose (x, y, θ) and a list of waypoints (X_i, Y_i) and outputs velocities (v, ω) to the robot's low level controller. Since this trajectory controller is completely self-contained, it can be used in combination with different modules that export and import the correct pins.

Figure 4(a) shows an implementation where the waypoint controller receives input from a GPS and a waypoint planner and outputs velocities to a clodbuster robot. This configuration was used to navigate teams of robots in outdoor environments [CCG2005]. The waypoint controller (the exact same module) has also been used for some indoor demos in the GRASP Lab in the configuration shown in Figure 4(b). In this case, instead of a GPS, an overhead camera is used for localization: the robot is tagged with a colored blob and a color blob extractor is used to compute its position. The waypoint list is given by a user interface running in the ROCI Browser and the robot being controlled is a Segway. It is important to mention that some of the modules are running in different machines but, since the pin architecture is network transparent, the module programmers are insulated from this usage issue.

4.2 Scheduling and Data Translation: The Networked Robot

In this example, a robot searches for a colored blob and, upon spotting, it dynamically changes the schedule to track the target and sends an e-mail to a cellular phone. This will exemplify several ROCI features such as allocation of priorities, strongly-typed interfaces, multiple data views, and general modular development. Importantly, there is minimal coupling between individual modules at the system code level. In fact, many of these modules are used without change or recompilation in many other tasks (for example, the Blob Extractor from the previous example). From another point of view, any one module in this task may be replaced to reflect a different hardware platform

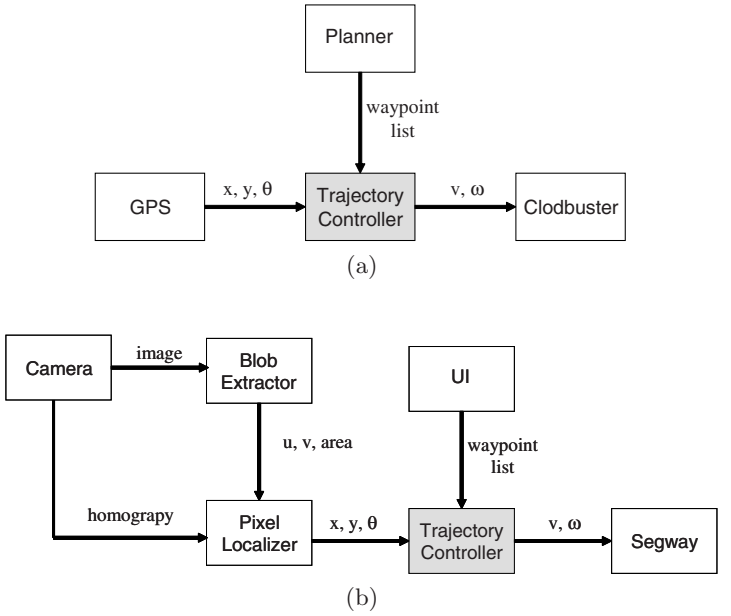

Fig. 4. Diagrams of two tasks using the same waypoint controller module: (a) outdoors with a GPS and a planner; (b) indoors with an overhead camera and an user interface.

or different functionality requirements without necessarily affecting any other module. Such robustness is achieved through the usage of the common, defined interfaces embodied by the ROCI pins in use.

A diagram of the relevant components (ROCI Modules) that make up this task is shown in Figure 5. Data is brought in from the outside world through a stereo camera module that interfaces, via native system code, with a camera driver. Color images from each camera are fed into blob extractors whose outputs are combined by a stereo blob localizer that computes the 3D position of the blob relative to the camera. This range and bearing information is fed into a motion controller that directs a mobile robot, a clodbuster, to follow the blob.

In this particular task, when no target has been sighted the robot pans its camera back and forth in a search mode. While in this state, the robot may be involved in processing data from other sensors or devoting a large portion of its CPU resources to long-term goals such as map building. When the blob extractors find a target, this data is fed to the blob localizer, but it is also fed to a schedule monitor module. This is the one module in this task that is specifically designed for one purpose: when a viable target is detected, processing resources are reallocated more heavily to the vision processing modules. In the actual experiment, the robot was able to process video data at approx-

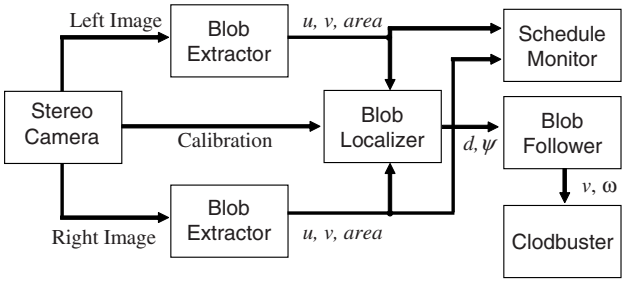

Fig. 5. Block diagram of modules involved in the blob following behavior.

imately 2.5 Hz while searching for a target. Once a target was detected, this rate jumped to 5 Hz, while the other CPU intensive processing jobs' rates slowed. This re-prioritization is effected by a change in the task schedule that specifies the relative rates each module iterates at. When the target is lost, the original schedule is replaced.

An additional feature of this experiment was an addition to the task file that specified that schedule change alerts should be messaged to a cell phone. This change did not require any change to the code for the modules or pins involved in the task. Instead, a single line of XML caused the system to automatically translate the internal binary data to a human-readable message, with labeled data, that was then sent to the cell phone at the number specified in the XML. In other experiments, robot status messages were forwarded to mobile devices. These status messages actually included the most recent views of a particular target being tracked. Such images appear in the status message as hyperlinks which, when followed, lead to images served up by the robot itself via HTTP.

5 Conclusion

The single theme that runs through all core ROCI development is the minimization of redundancy. In many cases, this involves the automation of as much programming effort as possible through the use of pervasive metadata. Since the system can always inspect what the programmer has done, it can generate code, invoke methods, and, in general, take action upon recognition of supported design patterns as simple as pin type declarations. This type of proactive system frees the developer from having to write the same glue code more than once, thus increasing efficiency and decreasing the likelihood of programmer error.

By focussing on the needs of the programmer, we have attempted to make component development as natural and clean as possible. As a result of freeing the developer from work that can be automated, and assisting in programmer cooperation through the use of self-describing objects, we are offering a system

wherein the lone developer may concentrate on the particular problem he or she wishes to solve. The enabling technology here is the modularity of development. Each developer may contribute his or her own high-level primitive to an already functioning system, and automatically gain from all existing works. In the end, the very early decision to define modules by their interfaces has lead all the way to truly high-level robot programming with all the features and support of a tightly integrated platform, and all the flexibility, specificity, relevancy, and robustness of ad hoc, loosely coupled, domain languages.

References

[Bru06] Brugali, D. and Brooks, A. and Cowley, A. and Côté, C. and Domnguez-Brito, A.C. and Létourneau, D. and Michaud, F. and Schlegel, C. *Trends in Component-Based Robotics*, In D.Brugali (Ed.) *Software Engineering for Experimental Robotics*, Spinger STAR series, 2006

[CCS2003] Chaimowicz, L., Cowley, A., Sabella, V. and Taylor, C. J. (2003) ROCI: A distributed framework for multi-robot perception and control. In *Proceedings of the 2003 IEEE/RJS International Conference on Intelligent Robots and Systems*, pp. 266–271, 2003.

[CCG2005] Chaimowicz, L., Cowley, A., Gomez-Ibanez, D., Grocholsky, B., Hsieh, M. A., Hsu, H., Keller, J. F., Kumar, V., Swaminathan, R., and Taylor, C. J.(2005) Deploying Air-Ground Multi-Robot Teams in Urban Environments. In *Proceedings of the 2005 International Workshop on Multi-Robot Systems*, pp. 223-234, 2005.

[CCT2006] Cowley, A., Chaimowicz, L. and Taylor, C. J. (2006) Design Minimailsm in Robotics Programming. *International Journal of Advanced Robotics Systems*. To appear, 2006.

[CHT2004] Cowley, A., Hsu, H. and Taylor, C. J. (2004) Distributed sensor databases for multi-robot teams. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, pp. 691–696 2004.

[CHT2004b] Cowley, A., Hsu, H. and Taylor, C. J. (2004) Modular programming techniques for distributed computing tasks. In *Proceedings of the 2004 Performance Metrics for Intelligent Systems (PerMIS) Workshop*, 2004.