Published in Proc. of the 1999 IEEE/RSJ International Conference on Robots and Systems (IROS'99), pp. 787-792, Kyongju, Korea, 1999

Controlling Image Processing: Providing Extensible, Run-time Configurable Functionality on Autonomous Robots

Tom Arbuckle Michael Beetz

University of Bonn, Dept. of Computer Science III,
Roemerstr. 164, D-53117 Bonn, Germany.
{arbuckle,beetz}@cs.uni-bonn.de

Abstract

The dynamic nature of autonomous robots' tasks requires that their image processing operations are tightly coupled to those actions within their control systems which require the visual information.

While there are many image processing libraries that provide the raw image processing functionality required for autonomous robot applications, these libraries do not provide the additional functionality necessary for transparently binding image processing operations within a robot's control system. In particular, such libraries lack facilities for process scheduling, sequencing, concurrent execution and resource management.

This paper describes the design and implementation of an enabling extensible system – RECIPE – for providing image processing functionality in a form that is convenient for robot control together with concrete implementation examples.

1 Introduction

Modern autonomous robots are faced with an increasing need for performing multiple, complex and changing image processing (IP) tasks. While the processing of visual data is resource-intensive (CPU, memory), contrastingly an autonomous robot may have only few resources available.

We can offset this high cost for performing IP by tightly integrating the IP to be performed within the robot's control system [1, 2]. One approach to this integration is to delegate the IP work to an image processing server.

In this paper we will describe a programming framework for the systematic construction of modular components of just such an image server, explaining the relationship of this framework to other architectures and systems in control and in image processing. We further illustrate, by means of concrete examples, how this system can benefit researchers seeking to satisfy both the requirements of image processing and those of autonomous robot control. By showing how this framework can systematically bind image processing functionality in a form tailored to robotic control, we both prove the need for such a system and provide a means of satisfying it.

1.1 Context – Related Work

This research straddles the topic areas of (autonomous) robot control and image processing. It is necessary to place our system - which we have called RECIPE - in context: we need to explain what our system is and what it is not.

There are many frameworks and architectures in the literature for creating autonomous robot control systems. Some systems, such as ORCCAD[3] or ControlShell[4], concern systematic ways to build software to control robotic hardware particularly with regard to real-time constraints. Other programming tools, such as RAP[5] or RPL[6], are designed to provide reasoning and planning capabilities operating on much longer timescales. RECIPE provides a *framework* for generating code for *controlling* IP and IP related activities under the *guidance* (online synthesis, scripting) of a reasoning and planning system.

Likewise, there are very many good IP frameworks freely available [7, 8, 9, 10] with a wide variety of abilities and features and, in addition, most IP research groups have their own home-grown software. RECIPE provides a platform neutral *framework* for simply and generically encapsulating *IP algorithms* together with mechanisms for capture and distribution of image data.

Integrating IP within the robot control system has convincing benefits. The control system has a great quantity of contextual information that can be used to guide the image processing routines. In addition, the control system is aware of the time and processing resources currently available to the robot and can tailor the IP to be performed accordingly. We require any such integration to be:-

1. **extensible** and **open** so that we, and others, can easily add functionality;

2. **programmable** and **run-time configurable** so that the operations performed can be varied according to the needs of the controlling system and;

3. **robust** so that problems with IP do not propagate to higher levels of the robot's control structure.

RECIPE provides a *framework* that satisfies these requirements flexibly and efficiently.

One earlier solution to merging IP within a control system as a reconfigurable IP server is the Gargoyle system[11]. RECIPE shares a similar underlying approach: the desire to make IP algorithms available to the control system as operators. However, perhaps the clearest analogue to RECIPE in the literature can be found in the work by Alami *et al.*[12]. They describe an architecture and programming tools which generate three-layered control systems: a decision layer, an execution layer and a functional layer. Their functional layer does not include decision capabilities or predefined behaviours and controls the interaction between the robot and its environment as a distributed real-time system. Objects at the functional level are modules that have a particular structure and are generated by a module generator.

RECIPE corresponds quite closely to the modules in the functional level of the Alami *et al.* architecture [12]. RECIPE provides a common generic and componentised *framework* for the generation of modules which are particularly (though not necessarily) concerned with image processing.

We note the similarity but stress that RECIPE modules have other capabilities which do not have direct correspondences in the Alami *et al.* architecture. By including scriptability and emphasising ease of creation of modules, we focus on providing a tool for programming IP for autonomous control systems.

In a previous paper[13], the twin to this one, we explained the control aspects of this union of IP and control. In this paper, we wish to elaborate on the programming framework that makes this integration possible.

1.2 Contents

In the remainder of this paper we will describe the overall design of the RECIPE system (section 2). The following two sections describe the structure and creation of RECIPE modules. We then describe the operation of the joint control and image processing system when carrying out image processing tasks (section 5). We close the paper by giving our conclusions, our plans for further work, some acknowledgements and the bibliography.

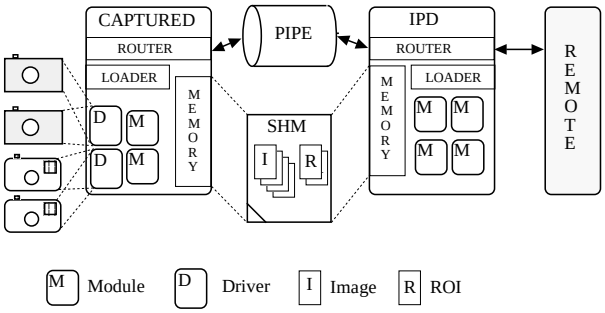

Figure 1: System Architecture. The process *captured* controls hardware and maintains the shared memory repository. The *ipd* processes provide users' execution shells.

2 RECIPE

RECIPE's architecture is shown in figure 1. RECIPE consists of two multi-threaded processes: a capture daemon *captured* and (possibly multiple copies of) an image processing shell *ipd*. An additional program called *cook* is provided for communication with the *ipd* process(es). In RECIPE, everything is a dynamically loaded active object [14]. All functionality, even the working parts of the applications, is provided as dynamically loaded and configured, modules with their own internal threads of control. The RECIPE framework provides facilities for sending messages to other modules, for loading and unloading modules, and for making large items of information, such as images, available to other modules. We now describe each of these applications in more detail, additionally commenting on the system's implementation.

2.1 The *captured* process.

This process forms the heart of RECIPE. It is responsible for image capture, data storage and for the restart and reconfiguration of failed *ipd* shells. It has three main components: a memory manager, a loader and a router. The memory manager is responsible for the management of data – such as images or regions of interest – which are placed in shared, persistent, memory. The second item is the loader which is responsible for the loading and unloading of RECIPE modules and drivers. Drivers are also RECIPE modules but they have the additional responsibility of managing the system's image capturing hardware, interposing a device independent interface between the rest of the system and the details of the hardware's operation. The third item is the router. When modules are loaded into the system, they register themselves with the router so that they may receive messages from other modules and from the outside world. A fourth component, a shell man-

ager (not yet fully implemented), records configuration and start-up messages from *ipd* shells to permit them to be restarted and reconfigured. Finally, the *captured* process communicates with *ipd* shells by way of pipes, one per *ipd* shell.

2.2 The *ipd* process(es).

Each *ipd* shell is a workspace where system users can employ the shared data made available by the *captured* process, loading and unloading modules to perform the actions the users require. The same components found in the *captured* process are reused here. The memory manager again provides access to the shared information and a loader and a router are employed for module loading and unloading and for message routing respectively. The *ipd* process transmits requests and some information to the *captured* process by way of the pipe it opens when it is created. Information from the outside world is handled by a special type of communications module which translates control messages from the form in which they are transmitted into RECIPE's internal message type. The *ipd* shells implement the interface between the control system and the modules which perform the image processing, distributing the control messages and managing the available computing resources.

2.3 The *cook* remote process(es).

The *cook* process (which runs remotely and not on the robot) is essentially an external version of the *ipd* shell. The main difference is that it provides means for the input of commands or scripts, forwarding them to the *ipd* shells by means of another copy of the communications module. Currently the *cook* process permits control scripts to be typed and sent to the *ipd* shells. Note that *cook* provides both a way of testing modules remotely and a means of driving the *ipd/captured* pair. In normal use, commands – delegated IP requests – come from the robot control system.

2.4 Implementation

RECIPE is implemented entirely in object-oriented C++ using the platform and compiler neutral framework ACE [15], the Adaptive Communications Environment, under constant development at the University of Washington. Many of the framework's components are repurposed as a result of the implementation. Additionally, the power and flexibility of ACE can be applied within the image processing modules.

Communication between the robot, or sensing station, is handled by a communications module. For compatibility with the other items of control software we run on our robot, this module currently employs the TCX message passing library [16] although a communications module using CORBA CDR encoding over a TCP/IP channel is currently under development.

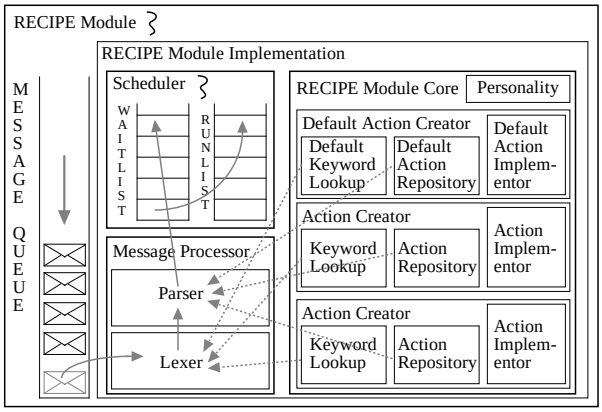

Figure 2: Module Architecture. All modules have at least two threads of control. Incoming messages are removed from a queue and parsed to produce runnable objects which are executed by a scheduler when necessary preconditions for their operation become true.

Here we point out one characteristic of the RECIPE system and design. The objects placed in shared memory are comparatively simple data containers. Unlike other image processing systems, RECIPE's purpose is the encapsulation and management of already existing functionality. Therefore a RECIPE image, for example, serves as a container for the visual data to permit it to be distributed throughout the system. Otherwise it provides only that information such as width, height and data format which will enable modules to uniquely decode the data.

3 RECIPE Modules – Structure

All user functionality is implemented in terms of modules. RECIPE modules are essentially multi-threaded shared objects having the structure shown in figure 2.

RECIPE modules are highly structured objects. The component visible to the system, labeled *RECIPE Module*, is charged with the delivery and acceptance of messages, containing instructions or data, which are distributed through the system by the processes' routers. Incoming messages are placed in a queue, labeled *Message Queue*. Messages are removed from the head of the queue by the *RECIPE Module*'s thread where they are passed to the *Message Processor*. The *Parser* and *Lexer* objects parse

instructions contained in the messages to create *Actions* to be carried out by the *Scheduler*. In particular, the *Lexer* employs the keywords in the *Keyword Lookup* section of the *Action Creators* to find the *Actions* held in the *Action Repository* objects.

Actions are in fact functions which manipulate the state of the *Implementor* objects. Once the *Actions* have been fully constructed, they are placed on a *wait list* where they are stored until any necessary pre-conditions for the execution of their functions are satisfied. Fulfilment of their pre-conditions permits the *scheduler*'s thread or threads to transfer the *Actions* to the *run queue* where the *scheduler* removes them from the queue and runs them. The additional *Personality* object is used to provide services, such as the module's name, that do not change during the lifetime of the module.

4 RECIPE Modules – Creation

As an example, we will show how to create a RECIPE module which carries out edge detection. To make the example concrete, we will set the task of making the function **edge_detect_filter** taken from the CVIPtools image processing library [17] into a schedulable, scriptable, image processing object within our framework. This CVIPtools function provides a variety of edge and line detection algorithms selected by its input arguments.

To create a RECIPE module with this functionality, we complete the following steps.

1. **Compile the library.**

2. **Encapsulate the required operations.** A prototype of the function to be invoked is added to the an encapsulating object's implementor as the interface between RECIPE and the CVIPtools library.

3. **Describe the operation to the system.** We declare the operation's name and arguments to the system. In our example, it makes a keyword, *edge_detect* for example, available to the module and tells the system about input and output parameters for the function.

4. **Name the module.** Choose a unique name.

5. **Create Actions.** Actions manipulate the implementor: they use the implementor as a tool to perform the processing operation.

6. **Create a Repository** All of the actions are collected in a repository where they can be accessed by the module's message processor. We need one action for each routine we want to invoke.

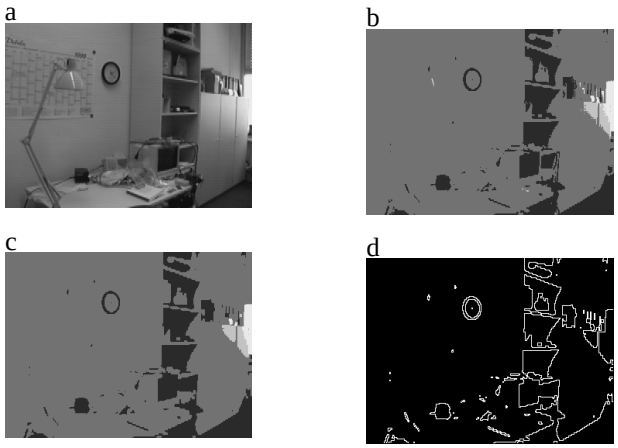

Figure 3: IP Operation Sequences. A script (text-file) is used to request sequences of operations which can, in principle, be performed in parallel.

7. **Image data format conversion routines.** CVIPtools expects CVIP-style images and RECIPE stores RECIPE_Images. We need to write conversion routines between one image type and the other. This implies that any reused code must be sufficiently open to allow this.

8. **Finished.** Using the conversion routines we have written, adding further functions from this library is simple. Total time to create a module with one function was under three hours, two of which were required to write data format conversion code.

Generic module code is very simple and repetitive: machine generation of modules is planned as an obvious extension.

5 Examples of Operation

We will provide three different examples of operation. Each has been chosen to illustrate different capabilities of the system. We also indicate how the operations performed mirror the requests of the controlling system.

5.1 Image Processing Scripts

One commonly requested function is the application of a sequence of image processing operations to an image captured from the robot's cameras. For example, if we suppose that the robot is attempting to localise itself using landmark features, sequences of IP operations selected to emphasise these landmarks can be requested by the controlling system. The sequence shown in figure 3 is typical: input (a), thresholding (b), grey-scale morphology (c) and edge-detection (d). These operations are all completely

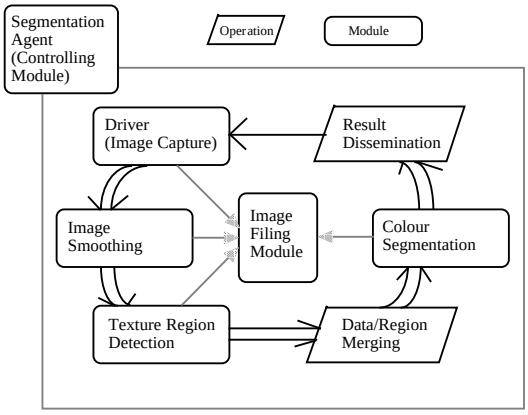

Figure 4: An IP agent controls and sequences the operations of five IP (tool) modules, exhibiting a primitive (autonomous) behaviour. The central module saves the images produced by the other four.

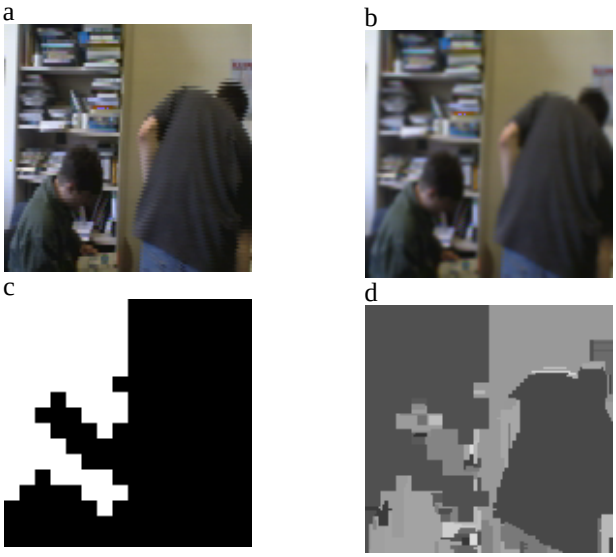

Figure 5: Agent's Output Images: input(a), smoothed input(b), textured region(c) and merged colour/texture segmentation(d).

standard and, in fact, are provided by a module which encapsulates all IP operations in the CVIPtools library. There are, however, two distinguishing features.

Firstly, these operations are performed by writing a *script*. The script is a text-file which contains sequences of commands in a simple language we have tentatively called *cuisine*. Each entry describes the action to be performed by which module and on which image. The second point is that, since RECIPE is intrinsically parallel in design, each operation can, in principle, be performed in parallel with any other. Writers of scripts are sometimes surprised when they find that the sequence of operations is not linear but must be controlled by introducing serialising synchronisation operations.

5.2 Higher level Modules – Agents

If a module passively provides a service or services, even if it is scriptable, we can think of it as a *tool*. Suppose we have some sequence of operations that we want to perform. This sequence is always the same but the parameters are different and varying continuously. Of course we could have the controlling system write a script for every invocation of this sequence and force it to provide the synchronisation and sequencing. An alternative, however, is to create a slightly higher level module, which we could call an *agent*, that itself controls the sequencing and parameterisation of the actions of other modules thus providing *behaviours*.

As an example, let us consider an agent for providing mixed colour and texture segmentation. The operations required of this agent are as shown in figure 4: image capture, smoothing, texture detection, merging and colour segmentation with output being saved at many stages thoughout. The *agent* controls and sequences the actions of four other *tools* with output as shown in figure 5.

The *agent* in this case, is providing a primitive form of (autonomous) *behaviour* which is configured, initiated, and adjusted according to context by the higher level controller. (One might say that such modules provide some of the supervisory layer functions of the Alami *et al.* architecture [12] discussed earlier.)

5.3 Modules with Interaction

Finally, we show a different class of module: a module which continuously analyses image streams and provides information on its results. This module is also an *agent* in that it is responsible for sequencing image capture (and possibly pan-tilt) requests and in that it also exhibits a primitive *behaviour*. The results of this module are shown in figure 6.

In this case, the module is continuously analysing an image capture stream to find simple pointing motions. As well as indicating successful detection of suitable features, it produces a target area where items of interest are likely to be located.

6 Conclusions and Future Work

In addition to the FAXBOT system [13, 18], also using RECIPE, in which we demonstrated the learning and reasoning aspects of combined control and IP, the above

a  b 

Figure 6: An Interactive *Agent*. Pointing gestures(a) in image sequences produce target areas(b).

examples provide a representative sample of the current image processing and manipulation abilities exhibited by the system (RECIPE 0.3.0). The scriptability, ease of construction of new modules and dynamic configuration have already proven valuable and we plan to exploit these features further. We will apply the system to: learning the IP to be performed from its environment; competitive comparison of IP algorithms depending on context; and research into means of deciding which algorithms to use and how they should be parameterised.

The RECIPE architecture itself presents many opportunities for further exploration. Given that it is almost a miniature operating system, there are opportunities for examining such issues as scheduling, the extent of scalability, shared-memory garbage collection and resource management. To conclude, future work will continue to reinforce RECIPE's position as a key component of the control software of our autonomous robot RHINO [19].

Acknowledgments

This work has been partially supported by the EU TMR Programme's VIRGO project (EC Contract No. ERBFMRX-CT96-0049). The authors would like to thank the reviewers for their helpful and constructive comments. One author (T.A.) wishes to thank Günter Kniesel for helpful discussions about object-oriented design. Gordon Kehren and Boris Trouvain have written or extended RECIPE modules, some of which provided data for this paper. Image segmentation modules are based on the work of Thomas Hofmann and Frank Eppink.

References

[1] D. Ballard, "Animate vision," *Artificial Intelligence*, vol. 48, pp. 57–86, 1991.

[2] I. Horswill, "Analysis of adaptation and environment," *Artificial Intelligence*, vol. 73, pp. 1–30, 1995.

[3] J.-J. Borrelly, E. Coste-Maniere, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro, "The ORCCAD architecture," *International Journal of Robotics Research*, vol. 17, pp. 338–59, April 1998.

[4] S. A. Schneider, V. W. Chen, G. Pardo-Castellote, and H. H. Wang, "ControlShell: a software architecture for complex electromechanical systems," *International Journal of Robotics Research*, vol. 17, pp. 360–82, April 1998.

[5] J. Firby, "Task networks for controlling continuous processes," in *AIPS-94* (K. Hammond, ed.), (Morgan Kaufmann), pp. 49–54, 1994.

[6] D. McDermott, "A reactive plan language," Research Report YALEU/DCS/RR-864, Yale University, 1991.

[7] G. Hager and K. Toyama, "X Vision: Combining image warping and geometric constraints for fast visual tracking," in *Computer Vision: ECCV '96 4th European Conference on Computer Vision* (B. Buxton and R. Cipolla, eds.), vol. 2 of 2, pp. 507–517, Springer-Verlag, April 1996.

[8] S. Smith and J. Brady, "SUSAN-a new approach to low level image processing," *International Journal of Computer Vision*, vol. 23, no. 1, pp. 45–78, 1997.

[9] K. Konstantinides and J. Rasure, "The Khoros software development environment for image an signal processing," *IEEE Transactions on Image Processing*, vol. 3, pp. 243–252, May 1994.

[10] A. Pope and D. Lowe, "Vista: a software environment for computer vision research," in *Proceedings 1994 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 768–772, IEEE Computer Society Press, June 1994.

[11] P. N. Prokopowicz, M. J. Swain, R. J. Firby, and R. E. Kahn, "GARGOYLE: An environment for real-time, context-sensitive active vision," in *Proc. of the Fourteenth National Conference on Artificial Intelligence*, pp. 930–937, 1996.

[12] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *International Journal of Robotics Research*, vol. 17, pp. 315–37, April 1998.

[13] M. Beetz, T. Arbuckle, A. Cremers, and M. Mann, "Transparent, flexible, and resource-adaptive image processing for autonomous service robots," in *Procs. of the 13th European Conference on Artificial Intelligence (ECAI-98)* (H. Prade, ed.), pp. 632–636, 1998.

[14] R. G. Lavender and D. C. Schmidt, "Active object: An object behavioral pattern for concurrent programming," in *Pattern Languages of Program Design 2* (J. M. Vlissides, J. O. Coplien, and N. L. Kerth, eds.), pp. 483–499, Addison-Wesley, 1996.

[15] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[16] C. Fedor, *TCX. An interprocess communication system for building robotic architectures. Programmer's guide to version 10.xx.* Carnegie Mellon University, Pittsburgh, PA 15213, 12 1993.

[17] S. E. Umbaugh, *Computer Vision and Image Processing: a Practical Approach Using CVIPtools*. Upper Saddle River, NJ 07458: Prentice Hall PTR, 1998.

[18] M. Beetz, "Structured reactive controllers — a computational model of everyday activity," in *Proceedings of the Third International Conference on Autonomous Agents* (O.Etzioni, J. Müller, and J. Bradshaw, eds.), pp. 228–235, ACM Press, 1999.

[19] S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Fröhlinghaus, D. Hennig, T. Hofmann, M. Krell, and T. Schmidt, "Map learning and high-speed navigation in RHINO," in *AI-based Mobile Robots: Case studies of successful robot systems* (D. Kortenkamp, R. Bonasso, and R. Murphy, eds.), Cambridge, MA: MIT Press, 1998.