Published in Proc. of the 1998 Computer Vision and Mobile Robotics Workshop, CVMR'98, pp. 91-98 Santorini, Greece, 1998

RECIPE – A System for Building Extensible, Run-time Configurable, Image Processing Systems

Tom Arbuckle, Michael Beetz *

University of Bonn, Dept. of Computer Science III,
Roemerstr. 164, D-53117 Bonn, Germany.

Abstract. This paper describes the design, and implementation of RECIPE, an extensible, run-time configurable, image capture and processing system specifically designed for use with robotic systems and currently under active development here at Bonn. Robotic systems, particularly autonomous robotic systems, present both challenges and opportunities to the implementors of their vision systems. On the one hand, robotic systems constrain the vision systems in terms of their available resources and in the specific form of the hardware to be employed. On the other hand, intelligent processes can employ sensory input to modify the image capture and image processing to fit the current context of the robot.
RECIPE meets these challenges while facilitating the modular development of efficient image processing operations. Implementing **all** of its functionality – within a platform and compiler neutral framework – as scriptable, active objects which are dynamically loaded at run-time, RECIPE provides a common basis for the development of image processing systems on robots. At the same time, it permits the image processing operations being employed by the robot system to be monitored and adjusted according to all of the sensory information available to the robot, encouraging the deployment of efficient, context specific, algorithms. Finally, it has been designed to encourage robust, fault-tolerant approaches to the action of image processing.

1 Introduction

Computer vision, in particular computer vision on robots, is a multidisciplinary research area which presents a different set of problems and criteria to systems designers than that exhibited by vision and image processing systems to be run on more general platforms. On the one hand, the robotic nature of the platform itself places many restrictions on the resources available to a vision system as well as requiring a much closer integration with hardware than is found generally. On the other hand, the mobile nature of the platform, the additional sensory information available to the robot and the practical nature of the tasks to be performed by most (if not all) robots, affords opportunities for optimisation and information integration that would not be generally applicable for desktop systems.

For example, consider the task of visually locating a bookshelf given the robot's approximate position within the room. The robot must apply image processing routines to the images captured by its cameras or other visual sensors to locate the object. There are several routines that can be applied to find a bookshelf which all have different resource requirements and different probabilities of success. By sensing its environment, the robot can select routines that are likely to give good results given its surroundings. Moreoever, it can evaluate its available computing resources and select a routine which gives the best chance of success in a given time period with respect to these resources.

Considering this example, our experience with the autonomous mobile robot RHINO [1] and a comparison of vision and image processing systems, we have identified several criteria which we believe are prerequisites for vision and image processing systems, particularly those systems to be deployed on autonomous robots.

- **Extensibility.** If the image processing software is to be adaptable, if the code written for the system is to be reused and if the code is to be employed for more than a few very specific tasks then we believe that the prime requirement for any imaging or image processing software (robotic or otherwise) is that it is both open and extensible. To be open and extensible, it must be modular and make as few assumptions as reasonably possible about its environment. It should also employ techniques and protocols which are themselves open and extensible.
- **Run-time configurability.** A close coupling between the robot control system and its sensory systems enables the robot's perceptual mechanisms to be tuned to its context [2, 3, 4]. Moreover, rather than having completely general algorithms that handle very wide ranges of parameters, an effective alternative strategy is to use less general procedures that only return good results with a high probability but also give indications of their failure [5, 6]. To employ this strategy implies that the visual systems of the robot must be able to be tailored to their

* Email: arbuckle,beetz@cs.uni-bonn.de

environment at run time. We take this to mean that the algorithms themselves can be replaced or run competitively against each other rather than the more simplistic interpretation of adjusting parameter values.
- **Stability.** The control system for a robot should be fault tolerant [7] and robust. Here image capture and processing are integrated with control. Thus the same requirements must be met by the imaging or image processing systems.
- **Resource Management.** The resources available to robots is usually limited. Memory, CPU time and communication bandwidth with the outside world are all constrained by the physical construction of the robot. The imaging system for a robot must be cognizant of these resource limitations and manage its use of them accordingly.
- **Platform independence.** To avoid duplication of labour and to strongly encourage code reuse, an imaging system for robots should be as platform independent as possible. It should be written in such a way that the same code can be used by merely recompiling it on a variety of operating systems and by a variety of compilers.
- **Hardware adaptability.** Much of the labour of fitting imaging systems to robots comes from the interface between the software and the hardware. Code has to be written which can drive the image capturing hardware and supply the visual information to the software. To be completely general, the system should be able to access information from general sensors [8, 9], not just CCD cameras and the like, as well as from arrays of visual sensors. Since it is impossible to accommodate all of these requirements automatically, any system should isolate the end user from the technical difficulties of driving the hardware but make it easy for system maintainers to add additional devices.
- **Image processing tools.** A powerful assortment of image processing tools and techniques which can be employed or aggregated to perform the image processing activities necessary for the robot's tasks is required by definition.
- **Symmetric multi-processing and threads.** Image processing systems and routines which have been written to take advantage of multiple processors can provide additional performance. Multi-threaded systems can take advantages of multiple CPU's if present at the cost of additional labour and some overhead (for locking).
- **Fast access to data.** Accessing the visual data should be convenient, easy and fast. Employing disk files (as a means of data sharing) or network or inter-process communication calls should be avoided if possible.
- **Standard library support.** The system should not preclude the use of other systems or standard image processing libraries. The system should permit the encapsulation of existing code, encouraging a high degree of code re-use.
- **Asynchronous operation.** The crucial (real-time) actions of collision avoidance and localisation require a high priority. Asynchronous operation of the system is very helpful. If the image processing can be done concurrently with other control processes, the results can be made available to the system when, and if, the system as a whole requires them. Conversely, the system is not preoccupied with image processing when outside events require its immediate attention.

In a sense our focus is on providing an extensible framework for image processing and image capture almost as an intermediary between the host computer's operating system and the robot's control system. This therefore makes the focus of this work fundamentally different from what is commonly thought of as an image processing or image understanding system. RECIPE is not only a system for the manipulation of images: it is a system which has both image processing and operating system like functionality; which manages resources on a users behalf; which asynchronously accepts and returns image processing information and which can also acquire sensory information including images. Where we have had a choice, we have chosen to emphasise functionality at the cost of additional labour. Although still a work in progress, the current system already comes close to meeting many of these requirements in full.

In the next section we describe RECIPE modules, the means by which RECIPE performs such tasks as image processing or communication. We discuss their function, their creation and use, and enumerate some very general prerequisites required for legacy software, such as image processing libraries, which is to be incorporated into the modules. Section 3 presents the RECIPE system. We subsequently describe the operation of the system as applied to a visual localisation task. After a section discussing related work, we list future plans and give our conclusions.

2 RECIPE Modules

Suppose that we wish to perform an operation such as Canny edge detection on an image. Further suppose that this operation is defined in an image processing library and that, rather than rewriting the code, we wish to reuse it as a RECIPE module. This scenario is used to illustrate the concept of RECIPE modules.

In RECIPE, all functionality is provided in the form of these modules. RECIPE modules are dynamically loadable active objects [10]. They are linked and configured at run-time, have their own thread(s) of control and contain a message queue as shown in figure 1. All threads have the same function. They are placed in a processing loop which waits on a condition variable for the arrival of messages from the rest of the system. (Message transfer will be explained later in the paper.) Once a message arrives, one thread awakens, removes the message from the queue, validates it and

parses it. Message parsing is currently rather simple. A command consists of text describing the module to carry out the operation, the operation to be performed and the arguments to be supplied to the operation. There is currently no restriction on what may be carried out by the thread. After function execution, the thread returns to waiting.

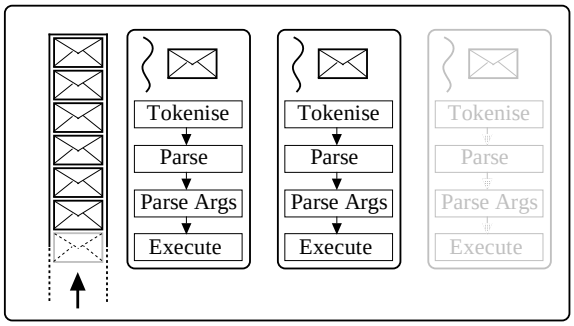

Fig. 1: Modules are active objects

We make the preliminary observation that RECIPE is entirely written in object-oriented C++. Thus to create a module which performs Canny edge detection using the library code, we derive a new class from one of several example classes – classes which are slightly specialised for different intended functions – provided in a library (libRECIPE). These classes provide default implementations of many required operations such as the message processing loop. We must define the names of the operations to be carried out by the module as well as functions which carry them out. We must also decide on the arguments to these functions. Facilities are provided for obtaining the values of the arguments to the module and for deciding on the operation to be invoked.

RECIPE stores image and region of interest (ROI) data as objects with character names. Given an object's name, the user can then request this object from a memory manager. Image objects are extremely simple containers for the image data. They merely store the image data and make it available in a raw form. To actually perform image processing on the data, as would be expected it must be transformed into a representation understood by an image processing library. For some libraries, such as the KUIM image processing library which uses a similar data representation, this can be done very quickly and easily. For others, which require a conversion step, it is better to maintain the image data in the form understood by the library, transforming it back again only after completing the processing.

There are a few restrictions on libraries that can be made into RECIPE modules or used within them.

- They should have a modular structure which clearly delineates the operations supplied.
- They must be compiled as thread-aware shared libraries. This simply means compiling with the correct compiler flags. However, libraries which are not thread-safe must also be protected by appropriate locking schemes unless it can be guaranteed that no more than one thread will execute within the library concurrently. (These are essentially the same restrictions placed on any multi-threaded code. Future module classes may provide this functionality for libraries which are known not to be thread-safe.)
- If they provide ROI structures, RECIPE can employ them although, again, some conversion may be required.
- Conversion between one image representation and another may be expensive so dynamic memory-based structures for intermediate storage are required. There may be colour-space issues for libraries which deal with colour. RECIPE is not currently aware of the colour space saved in its images but uses RGB for convenience.

The problem of conversion between data representations arises often when different image processing libraries are mixed (not to mention other problems such as name space polution). As RECIPE grows, modules designed to provide image conversion facilities – image processing servers if you like – will be written to take advantage of legacy code without format translation. Creating a Canny edge detection module was one of the first tasks to be carried out with the new framework. RECIPE is platform independent and run-time configurable. Current versions run under the Solaris and Linux operating systems with a variety of compilers. New modules can be added at any time by simply recompiling code and copying the resulting module to RECIPE's run-time directory. Thus RECIPE provides building blocks of image processing operations which can be chained together in pipelines and operate on shared images.

3 RECIPE

The design of the RECIPE system is shown in figure 2. RECIPE consists of two main applications – *captured* and (possibly multiple copies of) *ipd*. An application called *cook* is provided for communication with the *ipd* process(es) and for remote testing of modules. In RECIPE, everything is a dynamically loaded active object. All functionality, even the working parts of the applications, is provided as dynamically loaded and configured, multi-threaded modules. 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 modules in shared memory. We now describe each of these applications in more detail with the aim of showing how RECIPE meets the criteria listed in the previous section and additionally comment on the system's implementation.

3.1 The *captured* process.

This process forms the heart of the RECIPE system. 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 ROI's – which are placed in shared, persistent, memory. (This is currently implemented as a shared, memory-mapped, self-extending file.) The second item is the loader which is responsible for the loading and unloading of RECIPE modules – the software objects which perform the work of the system – 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. There is a fourth, and currently underdeveloped component (not shown) which receives configuration and start-up messages from *ipd* shells so that they can be restarted if the processes should crash. Finally, the *captured* process communicates with the *ipd* shells by way of named stream pipes, one per *ipd* shell. One new thread is spawned to handle each new connection in addition to those already employed by the *captured* process itself and by the modules.

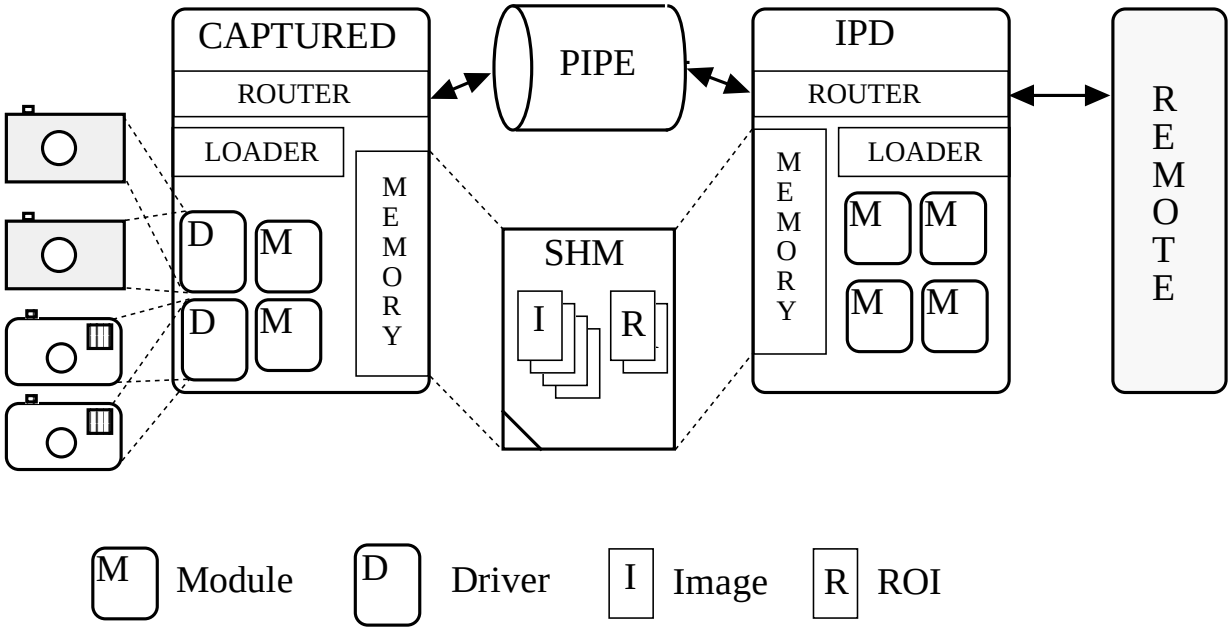

Figure 2: System Architecture

3.2 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. Its structure is similar to that of the *captured* process although somewhat simpler. The same components found in the *captured* process are reused here (although they may be configured differently). The memory manager, for instance, provides access to the shared information but is not responsible for the persistent storage in which the data are placed. Again a loader and a router are employed for module loading and unloading and for message routing respectively. The *ipd* process transmits configuration 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 messages which may be easily transmitted between modules by way of the routing mechanism.

3.3 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 differences are that it provides a different version of the shared memory manager (to permit the same modules to

be employed by simulating memory found in the *ipd* process) and 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. Future versions will probably incorporate some form of graphical front end for user convenience. Note that *cook* provides both a way of testing modules remotely and a means of driving the *ipd/captured* pair. In normal use, the command messages from the system can come from any kind of process such as, for example, a high level controller or indeed any process that can generate text messages.

3.4 Implementation

RECIPE is implemented using the object oriented, platform and compiler neutral framework ACE, the Adaptive Communications Environment, under constant development at the University of Washington [11] and already incorporated into several mission critical, commercial systems. Many of the framework's components are repurposed as a result of the implementation. The run-time dynamic linking is simply a light wrapper around the Service Configurator component of this framework and the modules are lightly disguised ACE Tasks communicating by way of ACE Message Blocks. The memory manager is also derived from the ACE Malloc class with a memory mapped file allocation pool.

Here we must mention one of the problems inherent in the RECIPE design and indeed any design that uses C++ with shared memory. Objects placed in shared memory may not have virtual functions (without trickery or special techniques). The use of C++ templates is possible (if the compiler you are using can compile them correctly). Thus RECIPE images and ROI's are perforce simple objects which act merely as data containers. More complicated representations may be used in the modules.

Communication between the robot, or sensing station, is handled by a communications module. Here at Bonn we have been using the TCX message passing library [12] as the basis for interprocess communications and RECIPE will use this method by default for conformity with our other robot control applications. However, TCX is now rather old and in many senses it is not suitable for a system claiming to use open standards. TCX is no longer under development or maintainance and it would be a very serious restriction on any system if it were forced to use it. Accordingly, we envisage that the communications module may eventually be replaced with one using a more current message passing protocol or perhaps with a CORBA based messaging service.

One question which might be asked is why RECIPE was not implemented in Java if we sought complete machine independence. Although there are some good arguments why Java could have been used for RECIPE, especially that part of RECIPE which interacts with humans, it was written in C++ for a combination of reasons based on speed of execution and coding methodology. We quote Stroustrup [13] saying: "For many applications, C++ is still the best choice when you don't want to compromise between elegance and efficiency."

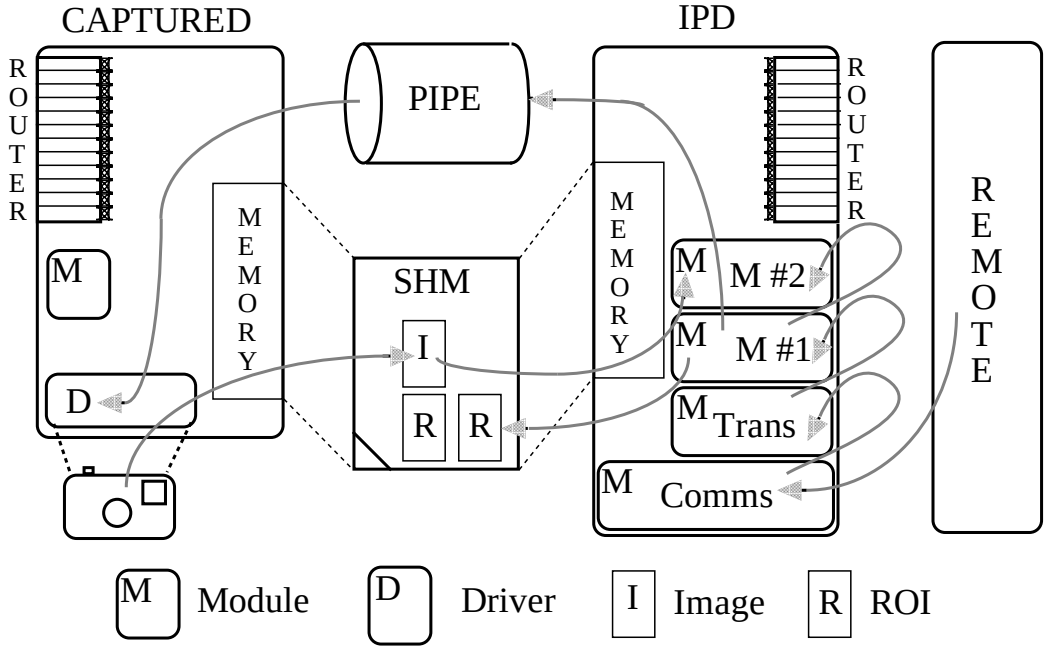

Figure 3: Object recognition task execution.

4 Operation and Results

Suppose that a robot is requested to visually verify its location. The robot is given (or has calculated) a reasonably accurate map of its environment. Verifying its location consists of finding and recognising landmark objects such as items of furniture known to be in its vicinity. We assume that a remote high level controlling process such as that found in the FAXBOT system [14] sends control messages to a RECIPE system running on the robot. We illustrate the sequence of operations involved in figures 3 and 4. (The example refers to version 0.2.0 of the software.)

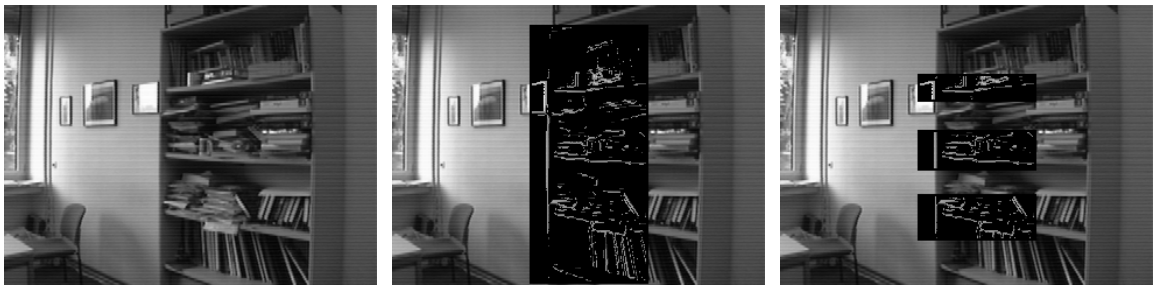

Figure 4a,b,c: Input, Canny output for 1 ROI, Canny output for 3 ROI's.

We assume that the robot (or rather the remote control system) knows that it should locate a bookshelf shown in figure 4a and that it also knows that its image sensors are pointed at the approximate location of the region. The first action required is the capture of the image to be processed. The remote system, which could be the *cook* application, sends an image request command to the *ipd* shell. This is received by the communication module which takes the plain text message transmitted over the network and packages it in a form suitable for transmission to other modules. The communication module then transfers this message to a translation module. The purpose of this module is to translate the textual form of the commands into that employed by RECIPE internally. This is to permit complete flexibility and can involve only a repackaging step if the message is already in the system's native format. The message is then transferred to the modules which are being used to perform the work in this case denoted M ♯1 and M ♯2. Since the command involves image capture, the module M ♯1 sends a command to the *captured* process by way of (one of) the connecting pipes and this message is then transfered to the driver module which controls the camera. The camera either creates an image in the shared memory and fills it with image data or takes a previously existing image from the shared memory and fills it. The image is then available for processing to modules in any of the *ipd* shells.

Further commands to define an ROI and to transform the image can be transmitted similarly with the module M ♯1 either performing the requested operations or possibly delegating them to another module such as M ♯2. The result of a Canny edge detection step destructively performed in a large ROI is shown in figure 4b. Since Canny edge detection is a rather expensive operation, the controlling system may decide to perform the operation only on regions of interest as in figure 4c or perform a different operation which may give similar results more quickly. The main point is that these operations are decided at run-time and that the modules to perform the work are loaded and unloaded as required.

5 Related Work

Image processing and image capture systems can be categorised as follows (although we note that there is some overlap between categories).

- **Systems for parallel computers.** These are general systems written to help their users take advantage of the additional power of these machines without having to be involved in the processing details Two examples are the PRIME [15] and CLONER [16] systems.
- **Systems for or using special hardware.** Such systems employ special image capture boards or special sensing apparatus such as camera arrays. The joint Improv-EyeBot system [17] couples an image processing system with a hardware platform for developing real-time imaging systems. HexEye [18] is described as a distributed optical proximity system. PROTEUS [19] is a multi-instruction, multi-data, parallel computer designed for image processing and image understanding tasks. There are also systems for employing multiple cameras[20, 21].
- **Systems for particular tasks.** The well-known X Vision system [22] developed at Yale provides several general tools for tracking image regions in images captured from a range of cameras under system control. The scene interpreter developed as part of the Vision as Process (VAP) project [23] and the SUSAN edge and corner detection system [24, 25] are two other examples. Specificity fosters good performance.

– **General systems.** These systems provide flexible and powerful image processing tools in a framework of general applicability and invite closest comparison with RECIPE. Particular examples are Khoros [26] and Advanced Khoros [27], TargetJR which is closely related to IUE [28, 29], Gargoyle [30], Vista [31] and the Windows specific MIL [32]. We should also mention the visual capabilities of such systems as Maple and Mathematica as well as general image processing libraries such as IPRS [33], KUIM, pbmplus and netpbm, and ImageMagick

The systems listed as general image processing systems require further discussion. The commercial MIL system [32], particular to Windows and to the Matrox imaging cards, provides means for capturing and manipulating captured images. The Vista system provides much functionality, including a user-interface toolkit, as a suite of interconnecting unix programs and user-defined data types. Arbitrary data types are important for seamless integration of the system with others. RECIPE provides means of transporting arbitrary data around the system but cannot easily provide arbitrary data types because of the restrictions placed on the data types by the shared memory implementation.

The Khoros system is a vast, extensible image processing system with many contributed modules and many types of functionality including a GUI based visual command language (Cantata [34]). Khoros is resource hungry and uses expensive mechanisms for communicating the visual information between the different modules. Some of the communication techniques to be introduced in Advanced Khoros - the use of CORBA, MPI and internet protocols - could be used to distribute the work to be performed by the system, given an adequate communications channel. Similar arguments about resource requirements can be applied to the TargetJr system – which will eventually merge with the IUE (Image Understanding Environment) – despite its use of dynamic loading for modules.

Finally there remains the Gargoyle system [30], a system with which RECIPE shares many similarities both in concept and in design. RECIPE is Gargoyle carried to its logical extreme. Gargoyle is a modular extensible image processing system developed at the University of Chicago. It uses Java modules calling C and C++ code to perform its image processing. It is multithreaded and run-time adaptable, receiving configuration commands from remote processes. RECIPE differs from Gargoyle in that RECIPE does not explicitly employ another system for image capture, employing a layering approach to isolate the system from the hardware requirements. RECIPE lacks Gargoyle's visual interface and is written entirely in C++ as opposed to Gargoyle's Java/C++ mix.

6 Conclusions and Future Work

Our main experience so far with the development of the system has been the value of the platform independence afforded by the ACE framework. The porting time from Solaris to Linux for the system was measured in hours. The same cannot be said, however, for porting between compilers. We make an empassioned plea for the C++ standard to be uniformly implemented by all compiler suppliers.

Concerning operation of the system, we are very encouraged by our early trials. The simplistic tests presented here represent only a fraction of the system's potential. The FAXBOT system shows how an image processing system can become an integral part of a robot control system. More research in this area is worth pursuing. When coupled with more image processing functionality incorporated from standard libraries or made available as utility modules, we believe that this system will become a core feature of our vision work on our robot (and hopefully those of others.)

There is always more that can be done. However, from the point of view of usability, some form of GUI front end, implemented as a module of course, and employing some form of dataflow [35] or pipeline metaphor is an item we feel should not be overlooked. There remain many practical areas - shared memory compacting garbage collection, auto-reconfiguration, image processing libraries, more device drivers, more compilers and operating systems including perhaps Windows NT - where work will continue. In conclusion, however, we believe that RECIPE has a great potential and we look forward to seeing it being used and extended by the robotics community.

Acknowledgement This work has been partially supported by EC Contract No. ERBFMRX-CT69-0049 under the TMR Programme.

References

1. J. Buhmann, W. Burgard, A. B. Cremers, D. Fox, T. Hofmann, F. Schneider, J. Strikos, and S. Thrun, "The mobile robot Rhino," *AI Magazine*, vol. 16, pp. 31–38, Summer 1995.
2. S. Ullman, "Visual routines," *Cognition*, pp. 97–159, 1984.
3. R. Bajcsy, "Active perception," *Proc. IEEE*, vol. 76, pp. 996–1005, 1988.
4. D. Ballard, "Animate vision," *Artificial Intelligence*, vol. 48, pp. 57–86, 1991.

5. B. Donald, “Planning multi-step error detection and recovery strategies,” *Int. J. Robotics Research*, vol. 9, pp. 3–60, February 1990.
6. S. Kannan and M. Blum, “Designing programs that check their work,” in *Proc. 1989 Symposium on Theory of Computing*, 1989.
7. J.-C. Laprie, J. Arlat, C. Béounes, and K. Kanoun, “Definition and analysis of hardware- and software-fault-tolerant architectures,” *IEEE Computer*, vol. 23, pp. 39–51, July 1990.
8. T. Okoshi, *Three Dimensional Imaging Techniques*. Academic Press, 1976.
9. J. Arnspang and B. Ruzena, eds., *1st ALCATECH Workshop*, (Sjaellands Odde, Denmark), July 1996.
10. 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.
11. D. Schmidt, “ACE: the Adaptive Communications Environment.” `http://www.cs.wustl.edu/~schmidt/ACE.html`.
12. C. Fedor, *TCX. An interprocess communication system for building robotic architectures. Programmer’s guide to version 10.xx*. Carnegie Mellon University, Pittsurgh, PA 15213, 12 1993.
13. B. Stroustrup and S. Hamilton, “The *real* Stroustrup interview,” *IEEE Computer*, vol. 31, pp. 110–114, June 1998.
14. 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.
15. R.-I. Taniguchi, Y. Makiyamna, N. Tsuruta, and S. Y. *et al.*, “Software platform for parallel image processing and computer vision,” *Proc. SPIE*, vol. 3166, pp. 2–10, July 1997.
16. J. Patel and L. Jamieson, “An object-oriented framework for the Cloner software prototyping environment,” in *Conference Record of 13th Asilomar Conference on Signals, Systems and Computers* (A. Singh, ed.), vol. 2 of 2, pp. 1354–1358, IEEE Comput. Soc. Press, 1996.
17. T. Braunl, “Improv and EyeBot real-time vision on-board mobile robots,” in *Proc. 4th Annual Conference on Mechatronics and Machine Vision in Practice*, pp. 131–135, IEEE Computer Society Press, September 1997.
18. S. Lee and J. Desai, “Implementation and evaluation of HexEye: a distributed optical proximity sensor system,” in *Proc. 1995 IEEE International Conference on Robotics and Automation*, vol. 3 of 3, pp. 2353–2360, May 1995.
19. R. Haralick, A. Somani, C. Wittenbrink, R. Johnson, and *et al.*, “Proteus: a reconfigurable computational network for computer vision,” *Machine Vision and Applications*, vol. 8, no. 2, pp. 85–100, 1995.
20. M. Frankel and J. Webb, “Design, implementation and performance of a scalable multi-camera interactive video capture system,” in *Proceedings CAMP ’95 Computer Architectures for Machine Perception* (V. Cantoni, L. Lombardi, M. Mosconi, M. Savini, and *et al.*, eds.), pp. 132–137, IEEE Computer Society Press, September 1995.
21. B. Batchelor, A. Jones, and P. Whelan, “Networks of intelligent multi-camera vision systems for industrial applications,” in *Machine Vision Appoications, Architectures and Systems Integration III*, vol. 2347 of *Proc. SPIE*, pp. 115–26, 1994.
22. 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.
23. P. Remagnino, J. Matas, J. Illingworth, and J. Kittler, “A scene interpretation module for an active vision system,” *Proc SPIE*, vol. 2056, pp. 98–107, 1993.
24. M. Perez and T. Dennis, “An adaptive implementation of the SUSAN method for image edge and feature detection,” in *Proceedings International Conference on Image Processing*, vol. 2 of 3, pp. 394–397, IEEE Computer Society, Cotober 1997.
25. 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.
26. 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.
27. J. Worley, M. Young, and L. Richards, “Advanced Khoros: the infrastructure of broad technology development,” in *Proceedings of the IEEE 1996 National Aerospace and Electronics Conference NABCON 1996*, vol. 2 of 2, pp. 474–480, May 1996.
28. C. Kohl and J. Mundy, “The development of the Image Understanding Environment,” in *Proceedings 1994 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 443–447, IEEE Computer Society Press, June 1994.
29. J. Mundy, “The Image Understanding Envirionment program,” *IEEE Expert*, vol. 10, pp. 64–73, Dec. 1995.
30. 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.
31. 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.
32. “MIL Matrox Imaging Library.” `http://www.matrox.com/imgweb/products/mil/mil.htm`, 1998.
33. T. Caelli, C. Dillon, E. Osman, and G. Krieger, “The IPRS image processing and pattern recognition system,” *Spatial Vision*, vol. 11, no. 1, pp. 107–115, 1997.
34. M. Young, D. Argiro, and S. Kubica, “Cantata: visual programming environment for the Khoros system,” *Computer Graphics*, vol. 29, pp. 22–24, May 1995.
35. C. Balasubramaniam, “Dataflow image processing,” *IEEE Computer*, pp. 82–84, November 1994.

This article was processed using the TEX macro package with SIRS98 style