# VIP: The Video Image Processing Framework Based on the MIRO Middleware

Hans Utz[1,2], Gerd Mayer[1], Ulrich Kaufmann[1], and Gerhard Kraetzschmar[3,4]

[1] University of Ulm, Neuroinformatics, Ulm, Germany
{gerd.mayer,ulrich.kaufmann}@uni-ulm.de
[2] now at Ames Research Center, Mountain View, CA, USA
hutz@mail.arc.nasa.gov
[3] Fraunhofer Institute for Autonomous Intelligent Systems, Sankt Augustin, Germany gerhard.kraetzschmar@ais.fraunhofer.de
[4] University of Applied Sciences Bonn-Rhein-Sieg, Sankt Augustin, Germany
gerhard.kraetzschmar@fh-bonn-rhein-sieg.de

## 1 Introduction

Application frameworks play a major role in fostering reusability of robotic software solutions. Frameworks foster the reuse of code and design and offer a convenient model of object-oriented extensibility. They provide a powerful concept for providing generic components of well-understood, modular solutions for robotics. Nevertheless, few such frameworks exist in robotics, especially at the application level.

Autonomous mobile robotics is a rapidly evolving field of research, and so far widely agreed upon standard solutions for subproblems have just begun to emerge. Furthermore, the substantial development effort required for developing and maintaining such an application framework has been avoided by the vast majority of research-driven projects, which tend to focus on the development of new ideas and concepts and the implementation of new algorithms.

When analyzing the situation in software development for autonomous mobile robots and the lack of reusable application-level infrastructure, one can identify two additional major obstacles on top of the issues already mentioned: Firstly, the hardware platforms used in autonomous mobile robots integrate a wide variety of different hardware, sensor, and actuator components, resulting in extreme heterogeneity. Secondly, mainly for efficiency reasons one can usually observe a tight coupling of software solutions to the low-level properties of sensory and actuatory devices. This coupling is likely to jeopardize, if not destroy, all high-level portability of robotic software solutions. Thus, as described in Chapter *Trends in Robotic Software Frameworks* it is much more difficult to make the higher development effort for a framework-based

D. Brugali (Ed.): Software Eng. for Experimental Robotics. STAR 30, pp. 325–344, 2007.

solution pay off in the long run by reusing the framework and its components for other scenarios or other robot platforms.

## 2 Middleware for Robots

Our research on software development in robotics addresses the intrinsic problems of this application domain and resulted in the design and implementation of MIRO, the "Middleware for Robots" [USEK02]. The basic idea of MIRO is to provide generic support for the management of the domain-specific difficulties by addressing them in form of a middleware-oriented architecture. For this purpose, MIRO provides several layers of functionality that reside between the bare operating systems and the robotics applications. The highest layer provides application frameworks for common tasks in robotics. The video image processing framework (VIP), which is the main subject of the discussion in this chapter, is part of this framework layer.

The frameworks make extensive use of the infrastructure provided by the middleware to ensure their portability, reusability, and scalability in long-term robotics projects. For these purposes MIRO provides a standardized distributed systems infrastructure, which was carefully configured and transparently extended in order to meet the requirements of multirobot teams, an advanced toolkit for configuration and parameter management, and generalized abstractions for sensor and actuator devices, which are modeled as network transparent services. So, before we discuss with VIP a particular framework in more detail, we will first survey some aspects of the lower layers of MIRO, in order to provide a clearer picture of their impact on framework-based development in robotics.

### 2.1 Communications Infrastructure

Robotics applications are inherently distributed. While this property is obvious in multirobot scenarios, single robots are also distributed systems. Many robot platforms are even equipped with multiple computational devices, encompassing embedded microcontrollers and PCs. They are often operated and monitored from remote workstations, or parts of the processing is offloaded to a powerful compute server. Therefore, we consider a distributed communication environment as an essential part of any robotics middleware.

The communication environment provided by MIRO is based on the CORBA standard. It provides a carefully configured, high performance communication environment [SGHP97] to the upper layers. System entities are therefore remotely accessible through object-oriented interfaces which are specified in the CORBA interface definition language (IDL). The design goal of the communications infrastructure provided by MIRO was to encapsulate

the complexity of the distributed system domain by a carefully designed default configuration, which is applicable to most robotics scenarios. Nevertheless, the underlying bare communications technology is still easily accessible to developers for special cases of adaptation.

Apart from the method call-based interfaces MIRO also provides event-based communication facilities, using the standard CORBA Notification Service [USM05]. This real-time oriented version of a publisher-subscriber protocol is transparently extended to meet the communication requirements of multirobot teams in wireless networks with heavily varying link quality [USM05]. Thus, MIRO not only offers a scalable, high performance communication infrastructure, but also allows to provide advanced features like generic logging facilities as discussed in [UMK04].

The CORBA technology offers mandatory features such as compile time type safety and programming language interoperability. Its high scalability and sophisticated co-location optimizations, automatically applied for objects that reside in the same address space, make up for most of the introduced computational overhead. Therefore, these interfaces are consistently used for local communication as well.

### 2.2 Configuration and Parameter Management

Software applications in robotics are full of parameters and magic numbers, such as the *maximum admissible velocity (with respect to the robot's current obstacle avoidance capabilities)* or the *encoder ticks conversion factor*, which allows to translate sensor measurements from the wheel encoders to a distance travelled by the wheels. Whenever better software revisions become available or outdated hardware gets replaced over the lifetime of an autonomous platform, these parameters tend to alter significantly. They also need to be adapted due to changes in the application scenario.

Hardcoding such parameters as constants might possibly allow for some optimizations by the compiler, but limits the flexibility of the design. Furthermore, these kinds of parameters exist throughout the entire robotic application software. Distributing these various parameters over dozens of files ruins quickly the maintainability of the system. Providing a central configuration header does little good either, as every change to this single file will result in an almost complete rebuild of the entire software package.

Simply switching to configuration files that are parsed at run-time results in other disadvantages. Firstly, it eliminates the static type safety provided by the compiler. Secondly, this approach requires every module to add parameter parsing facilities to its implementation for every single parameter that it provides. Also, configuration files introduce a new category of often subtle and hard to find bugs, such as mismatches of parameter names between the file and the parsing functionalities. Furthermore, configuration files usually just provide entries for parameters that differ from their defaults. So a separate

documentation has to be maintained, about which parameters are actually processed by the different subsystems.

To support robotics applications in this respect, MIRO provides a sophisticated design for configuration and parameter management. Parameters are specified in their own XML-based parameter description language. These specifications are then transformed by a compiler in classes of the target programming language (currently, C++ is supported). This code holds methods for reading and writing the specified parameters from and to configuration files. Furthermore, the specifications are also used by an interpreter to provide generic GUI-based editing facilities for the configurations files, that ensure type safety and syntactical correctness.

The design of this parameter toolkit not only keeps MIRO highly configurable and maintainable, employing this infrastructure on the framework layer actually propagates this features to the application layer.

### 2.3 Sensor and Actuator Services

Decoupling robotic algorithms from the physical capabilities of the robot requires to find suitable abstractions for its sensor and actuator devices. This task is the goal of the MIRO service layer. The basic idea is to allow programmers to write their software solutions against an abstract service specification instead of an individual physical device. The challenge in such generalizations is that many robotics applications need to address some of the unique individual properties of such devices in order to perform their task more effectively. Furthermore, a common interface will not abstract away physical differences between robot platforms such as the maximum admissible velocities.

The MIRO service layer offers strictly typed, network transparent, object oriented interfaces for sensor and actuator devices. Sensor devices not only provide method call-based interfaces, but also publish their data using the notification service. Interface hierarchies are used to solve the conflict between generalization of sensor/actuator modalities and access to the special features of an individual device. Meta-information can be queried from the services to make the properties of an individual device, such as e.g. the speed limits of motors, accessible to client programs.

The abstractions provided by the service layer provide an essential ingredient for framework-based development in robotics, as they enable the implementation of components reusable on multiple robot platforms. Furthermore, the frameworks provided by MIRO define network transparent high-level interfaces, introducing themselves as service to other subsystems of the application.

## 3 Application Frameworks

A good example of an application framework is VIP. VIP covers real-time oriented video image processing and is part of the highest layer of the robotics

middleware MIRO. It heavily draws upon the infrastructure for robotics application development discussed above. While a framework mainly provides a design scheme and components that help to meet the challenges of the problem domain (such as image processing), it also deploys infrastructure provided by middleware to support the developer in mastering the immanent challenges of autonomous mobile robotics.

First, the requirements for robotic vision systems are discussed, with a focus on highly dynamic and real-time constrained robotic scenarios, such as robot soccer, followed by a description of how the framework addresses these challenges. A hands-on example illustrates the concepts.

### 3.1 Video Image Processing

Vision systems for autonomous mobile robots must unify the requirements and demands of two very challenging disciplines: *i)* computer vision and image processing, and *ii)* robotics and embedded systems. While the state-of-the-art in computer vision algorithms is quite advanced, many computer vision methods are intrinsically computationally expensive. Even an efficient implementation of such methods cannot fix this problem. Therefore, the resource demands of computer vision methods are in conflict with the requirements posed by robotics and embedded systems. The latter demand very short execution cycles of the control loops that read out and process sensor data, interpret and fuse them, and determine appropriate actions for the actuators. Particularly, the real-time requirements of robotics seem to rule out most sophisticated computer vision methods, which is one of the reasons why some computer vision experts get discouraged to work on robot vision. As a consequence, robot vision software systems are often inflexible and hard to maintain, because they tend to contain hardcoded quick hacks, which for efficiency reasons try to exploit micro-optimizations like performing multiple operations simultaneously, or because they are heavily model-based or incorporate application-specific heuristics.

In order to mediate between the partially contradictory requirements of advanced vision processing in a real-time constraint environment, proper conceptual support for vision processing architectures is required. Such conceptual support should encapsulate the vision application within its application domain. For a better understanding of the different requirements that need to be supported, we briefly review some characteristics of the two problem domains.

#### Computer Vision and Image Processing

The basic concept of computer vision is the application of operators to image data, such as logical and arithmetical operations (conjunctions, multiplications), color conversions, morphological functions (erosion, dilation), filtering functions (Gaussian filters, convolutions), or linear transforms (Fourier or

wavelet transforms). Often operations transform more than one input image into a new output image. For example, the Canny edge detector [Can86] needs two images which are obtained by convolving a horizontal and a vertical Sobel operator respectively. Other operators, such as optical flow require as input a sequence of images obtained at different instances of time [HS80].In principle, an image operation can be viewed as a mapping of one or more input images into a new one.

More sophisticated operations cover more general input/output mappings, i.e. the result of an image processing operation does not have to be another image, but may be any other kind of data structure, such as a color histogram, a similarity measure between two images, or any other statistical measure on the image. In this case, the definition of a filter is extended to a mapping of one or multiple input images into a new image, or one or multiple classification values associated with the image.

Sequences of such image operators reveal features within the image that can be used to identify regions of interest (ROIs). Some filters do not work on the whole image, but only on parts of it. ROIs are used either to speed up the processing loop, or to make sure that the result of successive operations is not influenced or noisified by image areas which are already known to be irrelevant (e.g. in object classification). Various kinds of feedback loops, such as integration over time, can speed up processing and improve classification results [MMU05]. Because regions of interest can change between individual processing steps, they are associated to images just like the above mentioned classification values.

### Robot Vision

Performing operations such as those described in the previous section on the video image stream supplied by one or more cameras on an autonomous mobile robot imposes further constraints on the processing model:

Timeliness Constraints: Robots are situated in a physical world. For tasks like obstacle detection and object tracking the image processing operations must be calculated many times per second and preferably at full frame rate, which is typically 30Hz. The system designer needs to repeatedly assess the performance of the vision system and to ensure its efficiency. Whenever possible, image processing operations should be executed in parallel in order to fully exploit the available resources, such as dual-CPU boards, hyperthreading, and multicore processor technologies. More complex image operations, which need not be applied at full frame rate, should be executed asynchronously in order to ensure that the performance of other image evaluations is not negatively affected. Adequate processing models are required to support such designs.

Fixed Frame Rate Image Streams: New images usually arrive at a fixed frame rate. As the value of the obtained information rapidly decays in a dynamic environment, a sensor-triggered evaluation model is required.

Communication: Vision applications in robotics not only need to communicate their results to other subsystems of the application. For advanced image operations, information from other sensors needs to be accessed: Integration over time on a mobile robot requires access to the dead reckoning sensors of the robot. Visual object detection might be verified with range sensor readings, or even be matched with observations from other robots in an early fusion processing model.

Parameterization: Most image operators need to be parameterized in order to tune the quality of their results. Examples are the width of a Gaussian filter or the number of buckets for an color histogram. The calibration and optimization of parameters is an important part of the development process. Also, the configuration of the filter graph has to be altered frequently during development, which can be significantly facilitated by a flexible and configurable development environment for robot vision systems.

Development Model: Robot vision is performed on live image streams, which are recorded by a moving robot platform. This must be adequately addressed in the development model. For many vision applications, development starts on sets of test images and recorded test image streams. If the application domain implies nondeterminism, or if the robot's actions affect the quality of sensor data by inducing effects like motion blur, the vision system needs to be tested extensively in the real-world environment. This requires effective and stringent support for the test-debug cycle, for inspection, and for adaptation of parameters in the running application.

### Related Work

The widely known vision-related architectures can be roughly divided into three categories: subroutine libraries, command languages, and visual programming languages.

Subroutine libraries are by far the most widespread. They concentrate on the efficient implementation of image operators. Therefore, they typically provide a set of functions, each of which implements a different image processing operation. Well-known examples are the SPIDER system [TSTY83] and NAG's IPAL package [CCG89], which are written in C or Fortran. More recent libraries include LTI-Lib [sou05c] or VXL [sou05d], which are both open-source, written in C++, and provide a wide range of image operations covering image processing methods, visualization tools, and I/O functions. The commercial Intel Performance Primitives (IPP) [sou05b] are an example for highly optimized processing routines with a C API. What all these libraries lack is adequate support for flow control. Aside of yet another collection of mutex or semaphore helper classes, and possibly some kind of thread abstraction, there is no special flow control support available.

Command languages for image processing are commonly implemented as scriptable command line tools. In case of the imlib3d package [sou05a] the image processing operators can be called from the Unix command line. The

CVIPtools [Umb98] are delivered with an extended Tcl command language. Both packages provide programming constructs for conditionals and iteration. While the programmer has complete control over the system in a very flexible way, she also carries full liability over the dynamics of the image processing cycle. Additionally, the scripting-based approach does not make it any easier to meet the required performance constraints of typical robotics applications.

Visual programming languages are currently often viewed as the most sophisticated solution approach. They allow the user to connect a flowchart of the intended processing pipeline using the mouse. They combine the expressiveness and the flexibility of both libraries and command languages. Often, they provide not only a wide spectrum of image processing functions and statistical tools, but also a complete integrated development environment. Many of the available systems are commercial products, with VisiQuest (formerly known as Khoros/Cantata) being one of the most advanced ones. According to the information available from their web site, it supports distributed computing capabilities for deploying applications across a heterogeneous network, data transport abstractions (file, mmap, stream, shared memory) for efficient data movement, and some basic utilities for memory allocation and data structure I/O.

To the best of our knowledge there is no image processing framework that combines all of the features described above, like processing parts of the filter tree on demand and in a flexible yet powerful way. Such a capability would make the system suitable for a wider range of image processing tasks, like active vision problems on autonomous mobile robots.

### 3.2 Solution Approach

The challenges robotics imposes on computer vision mostly affect the non-functional aspects of image processing, such as resource management, time constraints and prioritization. Abstractions for the control flow of the programming logic, supplied by a framework-oriented approach, allow for a clean separation of the two aspects. The vision programmer only implements the individual image operations (if not already available in form of a library) and specifies the data flow for the target application. The VIP framework then executes the implemented code as specified by the execution logic, ensuring proper prioritization and synchronization of image processing cascades. The infrastructure provided by the lower layers of MIRO is employed to help mastering the general difficulties of the robotics domain, such as robot platform independence, networked communication, and configuration.

The central base class of the VIP framework is the `Filter`. It denotes not only a (non-)linear image transformation function like a Sobel operator, but any kind of input-output mapping including advanced operations such as a neural classifier on image features. The input of a `Filter` is a set of images and their associated meta-information. It produces an output image

as well as associated, user-definable meta-information which is handed to the successor filters. Buffering of image data is provided by a `BufferManager` class, which allows asynchronous image access as necessary e.g. for the processing of consecutive images.

While the control flow is specified in a tree and evaluated in depth-first order, the data flow is much more flexibly organized as a directed acyclic filter graph (DAG). The VIP framework ensures the correct evaluation order.

### Robotics Support

The framework performs sensor triggered evaluation of filters to prevent excessive polling or context switching between waiting threads. In order to maximize performance in this highly time constrained environment, VIP keeps track of which filters are actually queried by client modules. Based on this connection management, dynamic graph pruning is performed to process only those filters currently required by the processing tree. If a client module connects to a new filter, the filter is guaranteed to be part of the processing tree as soon as the next image becomes available. The VIP framework also provides support for network transparent as well as co-location optimized access to images of the filter DAG. Additionally, multiple filters can be queried simultaneously for ensuring their synchronizity. For co-located image queries a shared memory-based approach is used with zero-copying, by using the aforementioned specialized buffer manager.

### Image Sources

Within the framework, a video `Device` is modeled as a specialized `Filter`. It forms the root node of a processing tree and a source node in the data flow graph. The framework supports various camera connections, such as BTTV, IEEE 1394 and USB cameras (and also a file-based device used for development and debugging as explained below), which exist as black box components for the VIP framework.

Support is also available for processing the image streams of multiple cameras in parallel. Each processing tree is executed within its own thread and is processed in parallel with other source nodes, while the data flow can stay connected, as required for stereo imaging. The framework ensures appropriate synchronization between the image streams in such cases. Note that, as the framework takes care of synchronization, developers do not need to worry about locking issues and the right usage of synchronization primitives.

Additional processing trees can be utilized to decouple time-consuming image operations (a slow path), which cannot be executed at full frame rate, from fast image evaluations, which must be performed at full frame rate in order to ensure reactivity to unexpected events [UKM05].

**Real-Time Constraint Image Processing**

As one of the dominant features of robot vision is the timeliness constraint, VIP integrates multiple concepts for real-time processing. Each processing tree can be executed with its own thread priority and scheduler choice, which is directly mapped on the OS-native process scheduler by the framework. This is necessary to minimize jitter and ensure correct prioritization, especially under high load situations. Additionally, detailed timing statistics are provided for each individual filter and each filter subtree. Different models for synchronization of filters between different processing trees can be used to either optimize synchronization of image sources (stereo vision) or minimize locking overhead and context switching between threads (slow/fast path processing).

### 3.3 Example Configuration

The previously described set of features provided by the VIP framework is best illustrated by a small example. Figure 1 describes the derivation of an edge image for a standard test image of computer vision. The original image is blurred using a Gaussian and then transformed into a gray image. A horizontal and vertical Sobel operator is applied next. In the last step, the Canny operator is applied. The screenshots are taken from the generic inspection tool. Meta-information is not provided by these simple filters. Data and control flow are illustrated in Figure 1. The thick, solid lines denote the data flow, while the dashed lines illustrate the control flow.

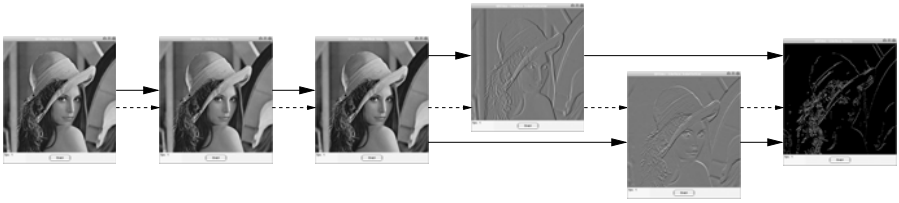

**Fig. 1.** Original image, intermediate processing steps (blurred, grayed and convolved images), and resulting edge detection. The thick, solid lines denote data flow, the thinner, dashed lines indicate control flow.

## 4 Middleware-Based Framework Development

The interaction of framework-based application development with infrastructure provided by middleware for robotics is best discussed using a small example. For this purpose, the task of writing a very primitive ball detector for RoboCup soccer robots is used. As the ball is red in this scenario, it is simply assumed that finding the center of all red pixel in the image will suffice.

```
<config_item name="BallDetection" parent="Video::Filter" final="true">
  <config_parameter name="redLow" type="int" default="200" />
  <config_parameter name="redHigh" type="int" default="255" />
  <config_parameter name="greenLow" type="int" default="0" />
   ...
  <config_parameter name="redRatio" type="double" default="0.001" />
</config_item>
```

**Fig. 2.** Definition of configuration parameters for the thresholds w.r.t. the colorspace dimensions and the overall threshold to decide if there is a ball or not.

This approach will admittedly not work on much more than the provided test image, but illustrates the concepts quite well. Object detection in highly dynamic scenarios with only limited control over illumination conditions requires far more elaborate processing as discussed in more detail in [MKKP05].

### 4.1 Parameter Definition

The ball detector will be implemented as a filter of the VIP framework. We expect the color of the ball to vary with illumination conditions. Also, even an image without a ball might have some red pixels by circumstance (shadows, reflections). Therefore, a threshold is necessary to decide when there is actually a ball in the image. To achieve high flexibility in the application of image operations, the VIP framework employs the parameter management facilities of MIRO. Therefore, we model these thresholds as MIRO configuration parameters and define them in the XML-based parameter description language provided. The parameter definition is shown in Figure 2. It specifies the different parameters with its respective data type and, potentially, a default value. In this example, the used parameter set contains only simple and plain values, but in general the parameter framework supports also much more complex, nested data structures and even single inheritance (seen here already by use of the `parent="Video::Filter"` attribute). Note that this Figure is a bit simplified for brevity and therefore omits some statements such as include directives.[5]

For debugging purposes the filter will provide an output image that shows all pixels classified as red (aka, belonging to the ball). The actual output of the filter is the center of the ball in image coordinates. In real-life robot soccer the coordinates of the ball would still have to be projected on the floor using the camera model in order to become meaningful. So, to make this data available for successive filters, the output is stored as meta-data of the output image. Meta-data is also defined in parameter definition language, as illustrated in

[5] The full source code of this example is part of the MIRO source code, which is available at `http://smart.informatik.uni-ulm.de/Miro`. For this example, see directory `Miro/examples/videoFilters/`

```
<config_item name="BallFeatureImage" parent="Video::FilterImage">
  <config_parameter name="x" type="int" default="-1" />
  <config_parameter name="y" type="int" default="-1" />
</config_item>
```

**Fig. 3.** Definition of meta-data using the MIRO parameter description language.

```
class BallDetection : public Video::Filter {
public:
  BallDetection(Miro::ImageFormatIDL const& _inputFormat);
  virtual void process();
};
```

**Fig. 4.** Filter class specification for a smple ball detection filter.

Figure 3. Again, this data structure is inherited from the appropriate base class `Video::FilterImage`).

The employment of parameter management facilities by the framework saves the developer coding effort, ensures the configurability of the implementation, and enhances its reusability. Furthermore, the infrastructure ensures the maintainability of the application by providing powerful facilities for parameter editing and run-time adaption.

### 4.2 Simple Filter Implementation

The VIP framework provides the `Video::Filter` base class for image operations. An actual filter implementation derives from this parent and only needs to specify the base parameters of the resulting output image and implement the actual image operation. The specification of the `BallDetection` is shown in Figure 4. In general, only the constructor for filter initialization and the method `process`, which is called to perform the actual image operation, needs to be implemented (another method called `init()` needs to be overwritten in more complex configurations as explained below). Furthermore, generic factory methods need to be provided for the parameter setting functionality discussed above. These are omitted in the example code of the figures.

Only the parameters of the output image need to be specified on construction of the filter, verifying that the input image has the appropriate data format. Our filter implementation expects the image to be coded in 24 bit RGB format and be of any size. The output image will have the same size as the input image (as assumed by default) but using gray scale format (see Figure 5).

The actual filter implementation is shown in Figure 6 for completeness. First a pointer to the image parameters (first three lines) as well as to the

```
BallDetection::BallDetection(Miro::ImageFormatIDL const& _inputFormat) :
  Video::Filter(_inputFormat) {
  if (inputFormat_.palette != Miro::RGB_24)
    throw Miro::Exception("Unsupported input format (RGB required).");
  outputFormat_.palette = Miro::GREY_8;
}
```

**Fig. 5.** Constructor implementation for the ball detection example filter. It simply checks the format of the incoming image and sets the appropriate output parameters like e.g. the color depth in this example.

```
void BallDetection::process() {
  FilterImageParameters * imageParams = outputBufferParameters();
  ImageParameters * output =
    dynamic_cast<ImageParameters *>(imageParams);
  unsigned char const * src = inputBuffer();
  unsigned char * dest = outputBuffer();

  int sum = 0, sumX = 0, sumY = 0;
  for (unsigned int x = 0; x < inputFormat_.width; ++x) {
    for (unsigned int y = 0; y < inputFormat_.height; ++y) {
      if ((src[y*inputFormat_.width*3+x*3] <= params_->redHigh) &&
          (src[y*inputFormat_.width*3+x*3] >= params_->redLow) &&
          (src[y*inputFormat_.width*3+x*3+1] <= params_->greenHigh) &&
          (src[y*inputFormat_.width*3+x*3+1] >= params_->greenLow) &&
          (src[y*inputFormat_.width*3+x*3+2] <= params_->blueHigh) &&
          (src[y*inputFormat_.width*3+x*3+2] >= params_->blueLow)) {
        sum++;
        sumX += x;
        sumY += y;
        dest[y*inputFormat_.width+x] = 255;
      } else {
        dest[y*inputFormat_.width+x] = 0;
      }
    }
  }
  if (sum > params_->redRatio * inputFormat_.width * inputFormat_.height) {
    output->x = sumX / sum;
    output->y = sumY / sum;
  }
}
```

**Fig. 6.** Process method implementation for the ball detection example. It includes image parameter query, input and output image requests, ball finding and output parameter creation.

input and output images (next two lines) are obtained. Then it iterates over all pixels of the image and calculates the average position of all red pixels on the x- and y-axis and counts the number of red pixels. If the number of red pixels lies above the parameterized threshold, the filter concludes, that there is a ball in the image and assumes it to be at the center of the red pixels. These coordinates are stored in the output image metaq-data (last two lines).

```
<instance type="Video::Parameters" name="BallDetection" >
  <parameter value="BallDetectionBroker" name="VideoBroker" />
  <parameter value="288" name="Width" />
  <parameter value="384" name="Height" />
  <parameter value="rgb" name="Palette" />
  <parameter name="Filter" >
    <parameter value="DeviceDummy" name="Name" />
    <parameter value="DeviceDummy" name="Type" />
    <parameter name="Successor" >
      <parameter name="Filter" >
        <parameter value="BallDetection" name="Name" />
        <parameter value="BallDetection" name="Type" />
      </parameter>
    </parameter>
  </parameter>
</instance>
```

**Fig. 7.** Filter tree configuration.

VIP provides standard filters for image acquisition from various types of cameras, as well as for reading stored images from disk. In order to put this simple ball detector to work all left to do is to combine it with an existing filter for image acquisition. Therefore it is necessary to introduce it to the VIP framework configuration machinery. For this purpose, a factory instance for this filter type needs to be added to the repository of existing filters of the VIP framework (Details on this step can be found in the complete source of this example).

### 4.3 Configuration

The configuration of the filter tree is provided to the VIP framework in form of an XML-based specification. A simple configuration is shown in Figure 7. Note, that values for the parameters specified in Section 2.2 can also be specified at this point. In this case we assume the defaults to be sufficient.

It is not necessary to write a single line of code for reading and parsing such an XML file, as this code is generated automatically by the use of the MIRO configuration parameter framework. The filter tree root node in this example is a file device filter, which reads images from disk to test the implemented algorithm offline. By simply exchanging the root node by a real device filter (such as e.g. a IEEE 1394 device filter), exactly the same XML file can be used on a real robot.

## 5 Development and Tool Support

The example developed so far demonstrated, how the VIP framework *minimizes the coding efforts* by providing generic functionality in the form of super

```
<instance type="Video::BallDetection" name="BallDetection" >
  <parameter value="true" name="InterfaceInstance" />
  <parameter name="Interface" >
    <parameter value="Ball" name="Name" />
  </parameter>
</instance>
```

**Fig. 8.** Interface definition for access to the output image.

classes and standard filters which can be easily customized. The infrastructure provided by MIRO allows for flexible configuration and parameterization. This enhances the portability of applications built upon VIP, as it e.g. allows to easily replace image acquisition filters, if the application would have to run with another type of camera on another robot.

This section shows how the MIRO infrastructure *supports the development process* and briefly sketches additional functionality which is targeted to ensure the scalability and maintainability of large scale image processing applications for robotics.

### 5.1 Interactive Inspection

Testing, debugging, and refinement of applications for autonomous mobile robotics requires specialized support, as monitoring the internal state of a robot in operation proves to be very difficult without altering the system under observation. For example, single-stepping through the code is not an option for a moving robot that needs to do reactive obstacle avoidance. For this purpose MIRO provides multiple concepts that support the development process. Firstly, the event-based communication can be generically written to persistent storage without further coding effort. Thereby streams of sensory and cognitive data from a robotics experiment can be logged to disc and replayed offline for inspection or statistical analysis. Secondly, the communication and configuration infrastructure allows to remotely query system information and alter parameter settings at run-time.

Each filter can also instantiate a service interface for network transparent access to its output image. This is exemplified in the specification of the filter parameters for the `BallDetection` filter in Figure 8. The interface is assigned an individual name to avoid ambiguities.

Figure 9 shows the result of the filter operation on a test image. It is visualized with `QtVideo`, a generic remote inspection tool provided by MIRO.

Additionally, the values of the parameters of each filter can also be inspected and altered at run-time, by the use of the configuration editor. Figure 10 shows the dialog for the parameters of the ball filter. The parameters of the filter can be adapted interactively at run-time, based on real-world data acquired by the robot in operation. Again, no code has to be written for all

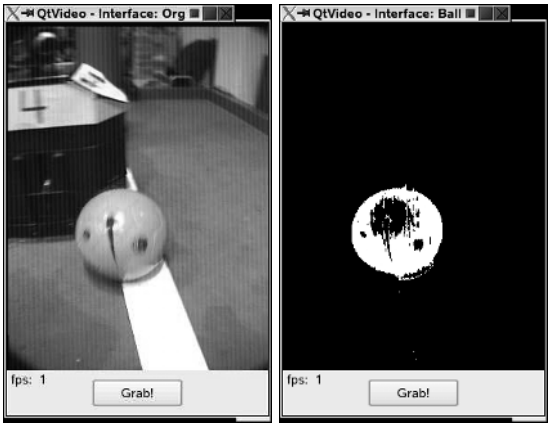


**Fig. 9.** Screenshots of the generic inspection tool QtVideo, showing an original image and the result of the BallDetection filter.


**Fig. 10.** Screenshot of the configuration editor, including the generically generated configuration dialog.

these tools. The parameter configuration is parsed at runtime to generate the appropriate configuration dialogs.

```
struct BallResult {
  long x;
  long y;
};
```

**Fig. 11.** Definition of the data structure in CORBA IDL, used for communication of the ball's position within the image.

```
EventChannel_var ec =
  _server.resolveName<EventChannel>("EventChannel");
supplier_ =
  new StructuredPushSupplier(ec.in(), _server.namingContextName.c_str());

StructuredPushSupplier::initStructuredEvent(event_,supplier_->domainName(),"BallResult");
supplier_->addOffer(event_);

...

BallResult result;
result.x = output->x;
result.y = output->y;
event_.remainder_of_body <<= result;
supplier_->sendEvent(event_);
```

**Fig. 12.** Example code to set up event communication via an event channel (upper part) and to finally fill and send the event itself (lower part).

### 5.2 Communication

So far, the result of the filter operation, i.e. the ball position in the image, is stored as part of the meta-data of the output image. With this method it is readily accessible for all successor filters. However, it would also be desirable to make this data available to other modules of the system. In our example, we choose to publish this data as an event, so that everybody interested can subscribe to this event and is automatically notified each time the ball filter processed an image.

For this purpose, a ball event is defined in CORBA IDL, as shown in Figure 11. The code required in the ball filter itself is also minimal, as illustrated in figure 12. The init() method of the filter is extended (upper part of the example). An event supplier is instantiated (first line), linked to the event channel of the robot (second line) and the event is initialized with the robots name and the chosen event name, for identification through the consumers (next two lines). At the end of the process method, the filter result is copied into the `BallEvent` data structure, and the data is copied into the payload of the event. Finally the event is sent to the event channel (lower part of Figure 12).

## 6 Application

The VIP framework is successfully applied in different robotic scenarios, such as biologically-motivated neural network learning, neural object classification in an office environment [FKSP04], and reliable high speed image processing in the RoboCup middle-size league [MKKP05]. The application of VIP in these very different scenarios resulted in the implementation of a large library of over 100 filters, which are shared between these projects.

Some statistics illustrate the complexity of these robot vision applications which are managed with the help of the VIP framework: The application in RoboCup consists of a dual camera setup, combining a directed camera for object detection, classification, and tracking with an omnidirectional camera for obstacle avoidance and near-distance ball tracking. The full RoboCup application currently consists of more than 120 filters with over 200 connections. One of the fastest path, a simple color-based football goal detection, takes only around 4 msecs to execute, while one of the slowest paths, a full, neural network-based classification of robots, requires around 20 msecs on average when seeing one robot per image (measured on 1.4GHz Pentium M).

## 7 Conclusions

Middleware-based application frameworks facilitate the reusability and scalability of robotics applications and help the development process.

In the first part of this chapter, the infrastructure provided by the Middleware for Robotics MIRO was introduced. Its lower layers, the communication and configuration infrastructure and the service based abstractions for robotics sensors and actuators, are nowadays also found in other robotics middleware architectures. Their implementation based upon open, mature standards such as CORBA and XML ensures highly robust and stable code and ensures a high amount of scalability.

A particular application framework of MIRO, VIP – the framework for video image processing, was introduced in the second part of the chapter. The design of the framework was discussed, based upon the requirements of advanced robotics vision applications in dynamic environments.

The third part of the chapter exemplified how the middleware provided infrastructure ensures reusability and maintainability of VIP-based vision applications and how the development process is supported by the infrastructure provided by MIRO.

The concepts of the application framework VIP discussed in the example are deliberately mostly not vision-specific, but are mandatory for robotics applications in general. Replicating them in every robotics application framework would not only result in considerable code bloat but furthermore introduce a significant development overhead. Furthermore, without middleware infrastructure like generalized sensor and actuator services, a robotics vision

framework could not provide advanced components such as information fusion with other sensory sources of the robot without loosing its portability. So the discussion of VIP in this section not only provides an introduction to application framework design in the context of autonomous mobile robotics, but exemplifies the necessity as well as the benefits of middleware-based application frameworks.

## References

[Can86] John F. Canny, *A computational approach to edge detection*, IEEE Transactions on Pattern Analysis and Machine Intelligence (1986), 679–698.

[CCG89] M. K. Carter, K. M. Crennell, E. Golton, R. Maybury, A. Bartlett, S. Hammarling, and R. Oldfield, *The design and implementation of a portable image processing algorithms library in fortran and c*, Proceedings of the 3rd IEE International Conference on Image Processing and its Applications, 1989, pp. 516–520.

[FKSP04] Rebecca Fay, Ulrich Kaufmann, Friedhelm Schwenker, and Gnther Palm, *Learning Object Recognition in a NeuroBotic System.*, 3rd Workshop on Self-Organization of AdaptiVE Behavior SOAVE 2004 (Horst-Michael Gro, Klaus Debes, and Hans-Joachim Bhme, eds.), VDI Verlag, Dsseldorf, 2004, pp. 198–209.

[HS80] B.K.P. Horn and B.G. Schunck, *Determining optical flow*, Tech. report, Massachusetts Institue of Technology, 1980.

[MKKP05] Gerd Mayer, Ulrich Kaufmann, Gerhard Kraetzschmar, and Gnther Palm, *Biomimetic neural learning for intelligent robots*, ch. Neural Robot Detection in RoboCup, Springer, Heidelberg, 2005.

[MMU05] Gerd Mayer, Jonas Melchert, Hans Utz, Gerhard Kraetzschmar, and Gnther Palm, *Neural object classification and tracking*, 4th Chapter Conference on Applied Cybernetics, IEEE Systems, Man and Cybernetics Society, 2005.

[SGHP97] Douglas C. Schmidt, Andy Gokhale, Tim Harrison, and Guru Parulkar, *A high-performance endsystem architecture for real-time CORBA*, IEEE Comm. Magazine **14** (1997), no. 2.

[sou05a] sourceforge, *imlib3d*, Available via http://imlib3d.sourceforge.net/, last visited 2005.

[sou05b] sourceforge, *Intel performance primitives (ipp)*, More information on http://www.intel.com/software/products/perflib/, last visited 2005.

[sou05c] sourceforge, *Lti-lib*, Available via http://ltilib.sourceforge.net/doc/, last visited 2005.

[sou05d] sourceforge, *Vxl*, Available via http://vxl.sourceforge.net/, last visited 2005.

[TSTY83] H. Tamura, S. Sakane, F. Tomita, and N. Yokoya, *Design and implementation of spider-a transportable image processing package*, Computer Vision, Graphics and Image Processing **23** (1983), no. 3, 273–294.

[UKM05] Hans Utz, Ulrich Kaufmann, and Gerd Mayer, *Advanced video image processing on autonomous mobile robots*, 19th International Joint Conference on Artificial Intelligence (IJCAI), August 2005, Workshop on Agents in Real-time and Dynamic Environments.

[Umb98] Scott E. Umbaugh, *Computer vision and image processing: A practical approach using cviptools*, Prentice Hall, June 1998.

[UMK04] Hans Utz, Gerd Mayer, and Gerhard Kraetzschmar, *Middleware logging facilities for experimentation and evaluation in robotics*, 27th German Conference on Artificial Intelligence (KI2004), Ulm, Germany, September 2004, Workshop on Methods and Technology for Empirical Evaluation of Multiagent Systems and Multirobot Teams.

[USEK02] Hans Utz, Stefan Sablatng, Stefan Enderle, and Gerhard K. Kraetzschmar, *Miro – middleware for mobile robot applications*, IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures **18** (2002), no. 4, 493–497.

[USM05] Hans Utz, Freek Stulp, and Arndt Mhlenfeld, *Sharing belief in teams of heterogeneous robots*, RoboCup 2004: Robot Soccer World Cup VIII (Berlin, Heidelberg, Germany) (D. Nardi, Riedmiller, Sammut M., and J C., Santos-Victor, eds.), Lecture Notes in Artificial Intelligence, vol. 3276, Springer-Verlag, 2005.