A Plug-and-Play Architecture for Cognitive Video Stream Analysis

Thor List, José Bins, Robert B. Fisher, David Tweed, University of Edinburgh

***Abstract*— This paper presents an architecture for cognitive analysis of streaming video, in which a new module can easily be plugged in, to add to or even compete with existing functionality. This allows the implementers to focus on the key scientific issues instead of struggling with the details of the implementation.**

The architecture is distributed and runs independently of the underlying computer architecture and can run transparently across one or many different operating systems in a larger distributed system. This architecture focuses on several key Computer Vision issues, such as multi-level global and local control, automatic dataflow based on auto-descriptive self-regulating independent modules that come together to form a whole based on the characteristics of the individual and the needs of the system rather than a static flow diagram.

***Index Terms*— Autocriticism, Autodescription, Autoregulation, Cognitive Architecture, Computer Vision, Modular Architecture.**

I. Introduction

IN this paper we will present the CAVIAR architecture for Cognitive Computer Vision, which allows the integration of many disparate modules into a system which performs cognitive analysis of video streams. Each module has complete plug-and-play functionality, using a straightforward interface that allows the global Controller to obtain detailed information about its functionality, its parameters and the quality and quantity of its results, and allows the Controller and other modules to make requests to alter its performance.

We describe the full architecture, which is based around a centralised Controller enforcing the global goals of the system of a large number of Modules. Each module functions as an independent unit, about which the Controller has no prior knowledge, but which will fully describe itself to the extent that the Controller knows exactly which function the Module performs including a full understanding of its parameters and its input and output.

Our model is based on previous work on process-based computer architectures [1], [2] with specific focus on dividing a system of modules into process federations where each has relatively autonomous control [3], although they participate in the system where the flow of data is not fixed [3].

The CAVIAR architecture has the ability to configure the system automatically at start-up as well as dynamically reconfigure parts or all of the system at runtime as needed. We will explain how each module describes its own capabilities and parameters, and how the Controller uses this information to become more robust to changes in the visual scene or when the system is running low on resources. Lastly, we will illustrate how the CAVIAR system can be distributed to utilise many computers, including those running a different operating system or even based on a different architecture.

Manuscript received November 26, 2004. Thor List, Robert B. Fisher, José Bins and David Tweed are with the Institute for Perception, Action and Behaviour at the University of Edinburgh in the UK. This research was supported by the CAVIAR project, funded by the EC's Information Society Technology's programme project IST 2001 37540.

II. Cognitive Vision System Architectures

Cognitive processes have to be very adaptive and able to deal with a great number of events and situations which cannot be fully specified at design time. Each process or module must be able to both be regulated individually and as part of a group, either by a local control unit or a global one with sufficient local knowledge.

Often a Cognitive Vision System will need access to many more modules than are needed at any one point in time, to cope with changes in the perceived environment or to its own internal state and goals. To perform in real-time all modules must be kept in an idle state waiting for a command to start processing data or at least be readily available when needed. A local or global control unit can then choose which modules would be most suitable for the current situation and even test several ones before making that determination. An advanced Controller could take the output from several modules, each performing the same task, and use all the information going forward.

A. *Global versus Distributed Control*

There are two main approaches to controlling a cognitive system [17]. Either one employs a global controller [16] which is aware of all parts of the system and has to know everything about them or one can use distributed control where sections of the system are controlled independently of the others [2]. Most systems use the global control approach as this is often easier and creates a more predictable application, although having a global controller makes scaling more difficult [2].

Distributed control is often hierarchical in nature, either divided into two levels with many local region controllers and a global controller interacting with these [15], or a many-level hierarchical structure where every few modules are controlled by one unit, which is itself controlled by a master of units, and so forth [2]. Even in the latter case there usually is an overall global controller that governs the goals and purposes of the system as a whole.

B. Global versus Distributed Dataflow

Dataflow regulation is of utmost importance in a Cognitive System as often the same data is used by several processes at the same time [17,18,19], and the modules downstream need the data in the correct order. This can either be done using a controller that will pass the data to the modules when available, or by a Blackboard-type architecture where modules subscribe to the data types or data stream needed and waits for these to become available [5].

Dataflow governs when which module can work on what dataset, and is in most cases determined by a combination of what specific modules need and what the overall goal of the system is. The dataflow can either be predetermined where the system designer dictates the exact path the data has to follow or dynamically adjusted to fulfil the system goals [15]. The latter can be split into two groups, namely globally versus locally determined flow.

Globally controlled dataflow means that a global controller with knowledge of the whole system knows who needs which data when and based on system goals makes changes to the flow of data as needed. This approach is the more popular approach when dealing with high volume media data streams, both because it is easier to manage, but also due to the inherent resource loads these streams can create when not managed correctly.

Distributed dataflow is often used with distributed control systems, where each module or group of modules determine dynamically which data they need when, and then request this data from the system as a whole. This is a cornerstone of the Blackboard-based architecture, which is excellent at handling discrete messages, but was found unable to scale or indeed handle the traffic efficiently when dealing with massive amounts of streaming data. Recently, a new type of Blackboard architecture named Psyclone was proposed [5] using Scheduling Whiteboards, which are Blackboards capable of handling both messages and media streams, and schedule subscriptions to both based on module and data priorities.

Although this approach was considered for the work presented in this paper, it was found that a global controller with globally controlled dataflow was more compatible with earlier work by the partners, and that a single controller could implement a very advanced dynamic rule and learning system, inspired by the work on ADORE [14] and VISIONS [19].

C. Distributed Systems

Most Cognitive Vision Systems require more resources than can be provided by one single computer, which means that strategic distribution of processes across a network of computers becomes very important. Other than the obvious issues of running different pieces of code on more than one computer and how to regulate this dynamically, one of the most crucial functions of such a system becomes making data available to the right module running on the right computer at the right time. Network delays can for the most part be predicted or at least anticipated, but the dataflow of such a system is vastly more complex and more resembles a chart organisation task than a simple flow diagram.

III. The CAVIAR Architecture

The CAVIAR Architecture uses one central Controller that knows about every module in the whole system, on both a global and local level. It controls the dataflow and schedules each module when the appropriate data is available. It regulates each module by either setting parameters directly or by requesting that the module regulate itself to achieve a certain goal, such as ‘find more objects’ or ‘output less features’. Each module obtains data from one or more sources and outputs one or more datasets plus feedback information. The feedback includes information on how well the module thought it performed and what it ideally would need the next time around, either for recalculating the current output or when the subsequent data becomes available.

A. The CAVIAR Controller

The CAVIAR Controller is written in a combination of Imalab Scheme [8], C++ and Clips, drawing on the strengths of all three and combining the versatility of Scheme with the efficiency and speed of C/C++ and the logic and efficient rule handling from Clips.

On start-up the Controller reads a minimal initialisation file containing a list of module names which can be used. Each module is instantiated and questioned for functionality, capability and for a full description of every parameter as well as the required inputs and datasets produced as output. From this list and provided with overall goals of the system the initial dataflow is calculated and the modules that come into play are initialised.

From then on the dataflow is auto-regulated based on the performance and feedback of the individual module and overall goals of the system. If a module produces a substandard response, it may be replaced by another module by inserting the new one in the place of the old and asking it to recompute the same data, or they may both continue to produce results and a split in the dataflow is created. A module with a substandard output will more frequently be asked to recompute with slightly different parameter values, set either directly by the Controller or indirectly by asking the module for more of one feature and less of another.

A more detailed description of the CAVIAR Controller can be found in [7].

B. The CAVIAR Modules

The CAVIAR Modules are all written in C/C++ for efficiency and draw on the combined resources of two libraries, the INRIA PrimaVision and CMLabs’ CoreLibrary, described in more detail in Section V (The CAVIAR Base Libraries). However, the CAVIAR modules are in no way restricted to only using these and frequently include other libraries, such as Intel's OpenCV and Clips.

The CAVIAR Modules operate on three basic principles, namely auto-description, auto-criticism and auto-regulation [2].

1) *Auto-description*

Each module provides a full description of its input, output and parameters when the Controller asks. The communication is handled transparently by the Base System (see Section IV) and the implementer describes the module in CVML [6] – an XML-based language extended for Computer Vision data streams.

```
<description>
   <parameters count="5">
       <parameter name="MaxGroups" type="integer" optional="no">
             <description>Maximum number of groups</description>
             <range from="1" to="100" step="5" />
             <default>10</default>
       </parameter>
       <parameter name="MinGroups" type="integer" optional="no">
             <description>Minimum number of groups</description>
             <range from="1" to="100" step="5" />
             <default>10</default>
       </parameter>
       .....
    </parameters>
    <dataflow>
       <inputs count="2">
            <input dataset="RawImage" />
            <input dataset="PointFeatures" />
       </inputs>
       <outputs count="2">
            <output dataset="Groups">
                <variable name="Time" type="Time" />
                <variable name="GroupList" type="GroupList" />
            </output>
            <output dataset="GroupHierarchy">
                <variable name="Time" type="Time" />
                <variable name="GroupPairs" type="GroupPairList" />
            </output>
       </outputs>
   </dataflow>
</description>
```

Fig. 1. An example of a full module description in CVML. This module takes many individual point features as input and groups them based on their spatial proximity.

Fig. 1 shows a full auto-description of a module that groups individual low-level image features previously computed by other modules. As input it takes two datasets, one called *RawImage* and one called *PointFeatures*. These datasets are defined by the modules that produce them and are required for this module to run so the controller needs to make sure that the modules producing these have run successfully.

The module produces two output datasets, one called *Groups* and one called *GroupHierarchy*. The *Groups* dataset has two variables; *Time* is a timestamp obtained from the *RawImage* and *GroupList* is a list of named groups of features found. These groups contain only the features and the next dataset contains information on the group hierarchy. The *GroupHierarchy* dataset contains two variables; *Time* which is the same timestamp as above and *GroupPairs* which is a list of statements $G_{id1} \subset G_{id2}$, indicating group containment relations.

The module description also specifies the module's parameters. Each has a type field, an optional field, a textual description, a default value and either a range of allowed values with a step size or a collection of discrete values, which the parameter can have.

The Controller uses these parameter specifications in two ways; for online control to regulate parameters by using rules during the normal system operation and for offline learning when comparing its own performance with training sets by exploring the whole or parts of the whole parameter space.

When executing the module has access to the input datasets it specified in its description. These are provided by the Base System and will be kept until the module and the Controller agree that this dataset is no longer needed. The module has full access to its own most recent variables and output (a number of steps back in time, can be specified in the auto-description) as well as its parameters that may have been set or changed by the Controller. Based on all of these the module can now compute its output datasets and provide feedback informing of its results.

2) *Auto-criticism*

High level feedback consists mainly of a quality and quantity measurement and the low level feedback can contain as much detail about the results as it wishes to inform the Controller about. As part of the feedback about its inputs it can also make requests to the Controller, such as *I need more of this and less of that* which the Controller then knows to relay to the module producing the relevant input dataset.

```
<feedback level="high">
   <result quality="0.98" quantity="2" />
   <run starttime="1101469301.262" endtime="1101469301.363" duration="101023"
         cpu="50140" user="30130" kernel="20010" />
</feedback>

<feedback level="low">
   <entity id="1" type="object" quality="0.98">
      <box x="10" y="12" h="45" w="58">
   </entity>
   <entity id="2" type="object" quality="0.98">
      <box x="10" y="12" h="45" w="58">
   </entity>
   <run starttime="1101469301.262" endtime="1101469301.363" duration="101023"
         cpu="50140" user="30130" kernel="20010" />
</feedback>
```

Fig. 2. An example both high and low level feedback from a module, which informs us that it is 98% confident about the two objects it found, along with information about when the module started and finished and microsecond details about how long and how much resources it took. The low-level feedback adds more detailed information about the tracked entities and their location using the CVML language [6].

An example of both high and low level feedback can be seen in Fig. 2. The high-level feedback contains information about the quality and the quantity of the output, which is 98% and 2, respectively. Also included is information about when the module did its processing and how much time and CPU resources it took. In this case the computation lasted slightly over 100 milliseconds of which 50 milliseconds of CPU time was spent.

The low-level feedback provides more detailed information about the results, such as the location of each of the tracked entities.

3) *Auto-regulation*

The Controller can choose to regulate the module by directly setting or changing the parameters described by the module. This is great for a system which has significant offline learning prior to running online, but for systems without much prior learning using the auto-regulation feature of a module is more robust.

Auto-regulation is part of the feedback protocol, where other modules or the Controller itself can make fuzzy requests. These usually take the form of wanting more or less of a specific feature or more generally wanting higher quality output. The module implementer knows best how to tweak the parameter to obtain the required result.

This is used extensively when the Controller wishes to either recompute one module's output until the needed quality or quantity level has been reached, or in the situation where a whole arm of the dataflow needs to be recomputed. Often this is also used to select between two equivalent modules.

IV. The CAVIAR Base System

The CAVIAR Base System consists of a code base which all the modules are based on and inherit from and which the Controller interfaces with. The Base System implements the Base Module, from which every module is derived and hence inherits all its functionality automatically. The Base Module and other support classes such as the Module Variables form the bulk of the Base System and are written purely in C++.

Each CAVIAR Module needs to have only two functions, *init()* and *compute()*, and to have its auto-description in the constructor. Everything else is handled transparently by the CAVIAR Base System, which provides the communication to the Controller and the other modules, and manages parameters, internal variables and the input and output datasets.

There are two sides to the Base System, the Controller interface and the Module Implementation interface. The former provides full support for everything the Controller needs from the module in terms of data I/O, access to parameters and feedback. The latter provides the modules with all the data and makes sure that everything is initialised.

A. *The Controller Interface*

The Controller communicates with the modules via an API specifically designed for this purpose. As seen in Fig. 3 the Base Module holds all of the data, feedback and parameters and merely allows the Controller and the Module Implementation to access these.

The Controller can query and set the values of existing parameters of several different types, such as integer, floating points, strings, vectors and more. It can enter requests and obtain feedback, redirect the input and output, and save the current or restore previous states in preparation for (re)computing results using the command interface.

The communication between the Controller and the modules happens either directly by calling the API functions or transparently via messages using the network. See the Section VI (*The CAVIAR Communication*) for more detail.

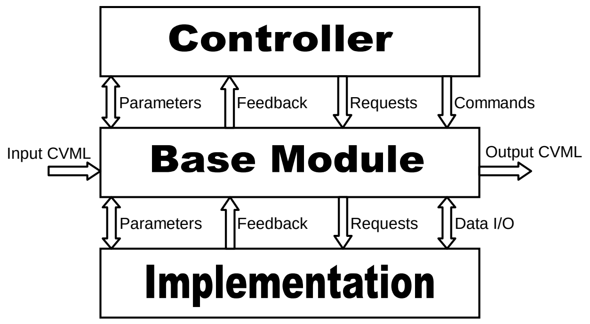

Fig. 3. A diagram of the Controller and Implementation interfaces to the Base Module. The input datasets (CVML) come into the Base Module from the left and when the Controller commands the module to run the module will compute the output datasets (also CVML), which are seen coming out on the right. The Controller can vary parameters, make requests and obtain feedback which the module implementation produces.

B. *The Implementation Interface*

Fig. 3 also shows that the Implementation Layer has a very similar access to the Base Module, in that it can read and set parameters, handle requests and provide feedback.

Instead of the command interface it has a generic Data API, from which it can obtain any data from previous runs (usually limited by the Controller to between 20 and 50) and access the current input datasets which it requested in its description. Using all this information it computes the output datasets required of it, which it marks as output using the Data API.

The high-level feedback is usually a few assessments of the quality and quantity of the results, but the low-level feedback can be as detailed as the module wants. For example, if the module is an object tracker and it found 20 objects in the scene, of which it is quite sure about 10, it may produce high-level feedback with quantity = 20 and quality = 0.5 and in the low-level feedback provide detailed information about each target including how confident it is about each of them. This may help an intelligent controller to determine how reliable the result is or how closely it matches the expected result.

C. *The Base Module*

The Base Module in Fig. 3 is the main handler of all the data, parameters, requests, feedback and commands, which it will route to the appropriate destination. The Controller will tell the Base Module from where it should obtain its input data, which could be one or more files or could be a network source.

Likewise, the Controller will command that the output datasets are either saved to a file or made available to other modules on request. The conversion of datasets to and from the common readable CVML [6] is handled transparently.

Every individual run has a unique ID and when the Controller wants to recompute either one or a whole sequence of runs, the module will handle the restoration of previous states, which it has stored in a buffer for safe keeping.

V. The CAVIAR Base Libraries

As mentioned earlier in this paper, the foundation of the CAVIAR system rests on two powerful libraries. The CAVIAR Base System is based on the CoreLibrary, which provides the independence of operating system and architecture needed. The PrimaVision library provides vision functions for the modules with full support for working with advanced vision.

The PrimaVision library [8] is part of the Imalab vision package and contains a full set of the basic functionality needed by any vision or video system, such as edge detection and optical flow. Also included is advanced functionality stemming from recent research by the INRIA vision group, such as object identification by elastic graph matching [11] and Gaussian derivatives [12].

The CMLabs CoreLibrary [13] is a multiplatform object library for C++ with transparent support for UNIX, Windows, Mac OSX and PocketPC. It provides many of the common objects found in Java (Strings, Collections, Queues, Math objects and built-in XML parsing) and has the ability to send objects and data across the network using Messages and Streaming Media. The network communication layer supports and autodetects additional protocols such as HTTP and telnet. Additionally, the CoreLibrary supports OS independent multi-threading including mutexes and semaphores.

VI. The CAVIAR Communication

The Controller chooses to run the modules in either a file- or network-based communication mode. The former is mainly used for offline testing or development, where every output is saved to a separate file and can be evaluated later. This mode also has the advantage that a single or a few modules can be run on the whole video set repeatedly in isolation, using a complete set of real inputs from previous runs.

The content of the dataflow is pure CVML which encodes both textual and binary information into an architecture independent data stream. This is very suitable for distributed systems, where the data has to travel across networks, but is equally suited for communication with other systems, which, even with no knowledge of the CAVIAR architecture, can use the information and even participate as a module, if required.

Network-based communication is used for online running or offline training. Each module auto-detects whether the intended recipient is in the same executable and can receive a direct in-memory transmission or it is located in a separate executable on the same or a different computer. If so, the data is automatically converted to CVML [6] and transmitted using TCP/IP communication. By converting to CVML it is ensured that even if the receiver is running on a completely different operating system or architecture that the data is still valid and understood.

VII. The CAVIAR Computer Vision System

The first implementation of the CAVIAR architecture was used for the CAVIAR project, funded by the EC as a collaboration between three institutes; Instituto Superior Tecnico in Lisbon, Portugal, Laboratoire GRAVIR-IMAG in Grenoble, France and Institute for Perception, Action and Behaviour in Edinburgh. The goal was to design a Computer Vision architecture which could observe human beings and analyse and predict their behaviour in common scenarios such as street scenes and shopping centres.

The CAVIAR architecture has been useful for the independent development of modules and controller for incremental construction of a combined system. Most modules were created and tested independently and when ready seamlessly integrated into the whole, either adding to the functionality or augmenting existing functionality. The Controller was also developed separately, initially working on mocked up modules and gradually integrating the real modules as well as increasing its own abilities to govern the system, first mostly based on rules, but later based on learning from training sets.

It took less than a week to set up the first version of the system, once the architecture was in place, and additional modules could be implemented in a few minutes, as they became available from the other teams.

When the Controller starts it creates an initial version of the dataflow from the auto-description of the modules. From this it decides which module needs to run when with which input. An example of an automatically constructed partial dataflow is shown if Fig. 4.

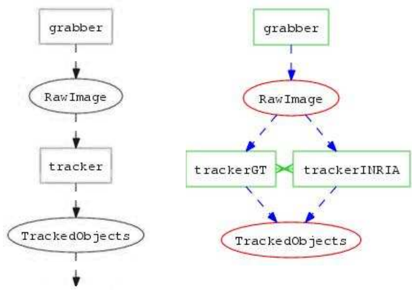

Fig. 4. Initial (partial) dataflow example, created by the Controller from the auto-description provided by the modules themselves. Example of dataflow for two equivalent modules which both take the same dataset RawImage as input and both produce the same dataset TrackedObjects as output. The equivalence is shown by the crossover lines connecting them.

The square boxes denote modules and the round objects are datasets. In this example, the *Grabber* retrieves the next image from the source (a camera or an offline file) and produces a *RawImage* dataset, which includes the image data and a timestamp. This dataset is requested by the *Tracker* which in turn produces a *TrackedObjects* dataset to be used by any modules who requests this dataset.

In the event that two modules produce the same dataset the Controller will consider these competitors and will see which produces the best output, either simply based on the feedback of the module itself, based on predetermined or learned rules or will even retry the computation of both until one wins or they both agree. A visual graph example of this is seen in Fig. 4, where two modules produce the *TrackedObjects* dataset and the Controller displays them as equivalent by the crossover lines connecting them.

Often, some parts of a dataset are used by some modules and other parts by others modules. To optimise performance and efficiency a dataset can then be split into two or more streams and modules can request one or more of these. An example of this is seen in Fig. 5.

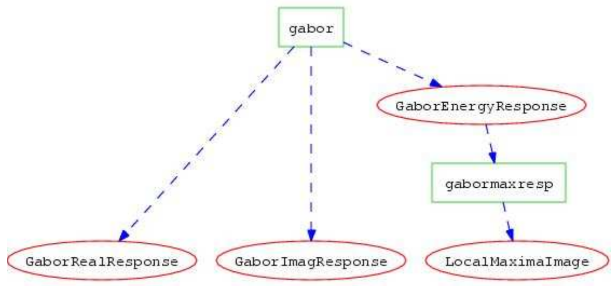

Fig. 5. Example of dataflow where one module outputs several different datasets, each needed by one or more other modules.

VIII. CONCLUSION

We have presented a plug-and-play Cognitive Vision Architecture called CAVIAR for analysing video streams. It has a global controller with full knowledge of the system as a whole as well as of the individual modules. Each of these are auto-descriptive and the system dataflow is automatically computed based on these and adjusted based on feedback from the modules and on the overall goals and performance required from the system. The modules are auto-regulative in that the Controller or other modules can request that it modifies its own parameters to achieve a desired output. And each module is auto-critical as they continuously evaluate their own performance and report this back to the Controller. Finally, we presented the CAVIAR Computer Vision System which based on this architecture allowed us to implement a fully working cognitive vision system in less than a week, and add new modules in only a few minutes.

REFERENCES

[1] A. Finkelstein, J. Kramer and B. Nuseibeh, “Software Process Modeling and Technology”, Research Studies Press, John Wiley and Sons Inc, 1994.
[2] J.L. Crowley, "Integration and control of reactive visual processes", Robotics and Autonomous Systems, vol. 16, pp. 17--27, 1995.
[3] J. L. Crowley and P. Reignier, “Dynamic Composition of Process Federations for Context Aware Perception of Human Activity”, International Conference on Integration of Knowledge Intensive Multi-Agent Systems, KIMAS'03, 2003.
[4] A. Caporossi, D. Hall, P. Reignier and J.L. Crowley, “Robust Visual Tracking from Dynamic Control of Processing”, PETS '04 - Performance and Evaluation of Tracking and Surveillance, 2004.
[5] K. Thórisson, C. Pennock, T. List, J. DiPirro, “Artificial Intelligence in Computer Graphics: A Constructionist Approach”, Computer Graphics Quarterly, Vol 38, Issue 1, pp 26-30, New York, February 2004.
[6] T. List, R. B. Fisher, “CVML - An XML-based Computer Vision Markup Language”, Proc. Int. Conf. on Pat. Rec., Cambridge, Vol 1, pp 789-792, 2004.
[7] J. Bins, T. List, R. B. Fisher, D. Tweed, “Task independent control of cognitive video sequence processing”, IEEE CAMP'05 Int. Workshop on Computer Architecture for Machine Perception, 4-6 July, 2005.
[8] A. Lux, “The Imalab Method for Vision Systems”, Machine Vision and Applications Vol. 16 No. 1, p. 21-26, 2004
[9] D. Hall and J.L. Crowley, “Computation of generic features for object classification”, Proc. Scale Space Methods in Computer Vision, Skye, UK, pp 744-756, June 2003.
[10] J.L. Crowley and O. Riff, “Fast Computation of Scale Normalised Gaussian Receptive Fields”, Proc. Scale Space Methods in Computer Vision, Skye, UK, pp 584-598., June 2003.
[11] M. Lades, J.C. Vorbruggen, J. Buhmann, J. Lange, C. von der Mahlsburg, R.P. Wurz and W. Konen, "Distortion Invariant Object Recognition in the Dynamic Link Architecture", Transactions on Computers, March 1993, Vol 42 Num 3, pp 300-311.
[12] F. Pelisson, D. Hall, O. Riff, J.L. Crowley, “Brand identification using Gaussian derivative histograms”, Proc. International Conference on Vision Systems, Graz, Austria, pp 492-501, Apr 2003.
[13] CMLabs CoreLibrary Homepage: http://www.cmlabs.com/corelibrary
[14] B. Draper, J. Bins and K. Baek. "ADORE: Adaptive Object Recognition," International Conference on Vision Systems, Las Palmas de Gran Canaria, Spain, Jan 13-15 1999. pp. 522-537.
[15] V. Bulitko, G. Lee, I. Levner, “Evolutionary Algorithms for Operator Selection in Vision”, Proc. FEA 2003 workshop, North Carolina, USA, September 2003.
[16] D. Crevier and R. Lepage, “Knowledge-based image understanding systems: a survey”, Computer Vision and Image Understanding, Vol 67 Issue 2, August 1997, pp 160-185.
[17] M. Thonnat, S. Moisan and M. Crubézy, “Experience in Integrating Image Processing Programs”, Proceedings of the First International Conference on Computer Vision Systems, 1999, pp 200-215.
[18] Matsuyama, T., and Hwang, V., “SIGMA: A Knowledge-Based Aerial Image Understanding System”, New York: Plenum, p. 277-296, 1990.
[19] Hanson, A.R., and Riseman, E.M., “The VISIONS Image-Understanding System”, Advances in Computer Vision (Ed. C. Brown), pp 1-114, 1988.