Computing and Informatics, Vol. 31, 2012, 1279–1293

SCRIPT LANGUAGE FOR IMAGE PROCESSING

Jiří Zuzaňák, Pavel Zemčík

Faculty of Information Technology
Brno University of Technology
Božetěchova 2
612 66 Brno, Czech Republic
e-mail: zuzanakjiri@gmail.com, zemcik@fit.vutbr.cz

Communicated by Dieter Kranzmüller

Abstract. This paper proposes a design and structure of script language which is intended for easy description and prototyping of high-level image processing operations. The image operations are meant to be composed from basic building blocks represented by either C/C++ functions or appropriate block connections in FPGA (Field-Programmable Gate Array) circuits. The proposed language is designed for use in systems for rapid prototyping and testing of image processing applications as well as for final implementations of the applications. The integration of language into such systems is discussed as well as explanations of parts of the image processing system as seen through the interface of the proposed scripting language. The paper targets structures and syntax of the language, parallelization of high-level image operations and communication between the multiple instances of interpreters of the scripts.

Keywords: Script language, interpreter, image processing, embedded systems, image operations, image processing pipeline, parallel computation

Mathematics Subject Classification 2010: 68U10, 68N15, 97P40

1 INTRODUCTION

Research in computer vision and image processing resulted in many standalone algorithms targeted to specific tasks. The algorithms can be categorized into several

classes. As an example of such classes, the edge detection algorithms, feature extraction algorithms, segmentation algorithms, object detection algorithms, etc. can be mentioned. The algorithms from these classes can be combined in order to achieve complete high-level image operation. This approach is also known as image processing chains (IPC). An example of simple IPC, namely the following chain can be introduced: grayscale transformation → LBP (local binary patterns) extraction → sparse histogram computation. This chain represents simple feature extractor based on LBP.

Most of the experimental tasks or computer vision (intended for testing of new algorithms) can be constructed by combination of such basic building blocks. Let us assume that we have available a library of such algorithms represented using C/C++ functions or equivalent FPGA blocks. Then we can use some high-level tool for description of their connections and possibly their parameter setting to get the desired functionality.

Short design time of application prototype (experimental framework, or computer vision application) is one of the essential requirements on implementation tools. Further requirements to be complied by the proposed system include: easy use, transparency to the parallelism and the optimization policy, efficiency (performance gain in most common image processing operations), portability, robustness (provided computation should be insensitive to variations of input data, and should provide correct results), completeness (it should not be necessary for user to use packages not related to the designed system). The proposed Script Language for Image Processing (SLIP) should perform the above mentioned task through description of high-level image operation stream. During the application creation process, the stream is described through definition of its node operations and by description of their connections. The stream is then preprocessed in order to recognize parallel computations, which can be exploited for faster execution of operations during the script execution.

This paper is organized as follows. In Section 2, a motivation for design and implementation of the proposed language – yet another scripting language – is described, in this case, the language for description of image operations. Section 3 describes the proposed script language in more detail and discusses its role in image processing system. Section 4 introduces classes used in SLIP designed especially for description of image processing operations and description of usage of these classes for description of particular operations. In Section 5, data stores designed for communication between scripts as well as other components of image processing system are described in detail. Finally, Section 6, concluding the paper, contains description of the achieved results and proposed ideas for further work.

1.1 Related Work

Description of Image Processing Chain (IPC) for digital still cameras is nicely done in [4] by A. Gentile, S. Vitabile, L. Verdoscia, and F. Sorbello. This work studies the IPC concept used for processing of data retrieved from digital still cameras.

The described IPC concept is enhanced for execution in SIMD Pixel Processor (SIMPil) architecture. The architecture is designed for image and video processing applications. Image processor based on SIMPil architecture is designed to meet real-time requirements in image and video processing. Direct coupling between I/O and processing elements allows computations performed on pixels, as data arrives in a stream-like fashion, rather than having data transported over centralized image buffer. IPC approach based on streamed data processing accelerated in hardware architecture is similar to that presented in this paper.

In [5] Rin-ichiro Taniguchi, Naoyuki Tsuruta, Shigeru Kusakabe and Makoto Amamiya describe, a functional language V that is designed to be independent from target architecture and to provide a framework to programmers to easily describe wide variety of parallel algorithms. Though language V is described as a general purpose language, its major application is identified as image processing, computer vision and multi-media processing. Parallelization in V is achieved by creating instances of agents from so called agent templates. If agent instances have no dependency among them, they are executed concurrently. As in the approach presented in this paper, the functionality connecting functional block is hidden under user friendly interface of the language.

In [9] Jon A. Webb introduces the Adapt language. Adapt is architecture independent language based on split and merge image processing model. The image is divided into portions and each image portion is processed independently. Separated results from computations performed on these portions are combined in final step of computation. The paper further examines the split and merge model and state that is capable of computing any image operation that can be computed in forward or reverse order over image data structure. The image is partitioned by rows; finer partitioning is useless, split and merge model does not make enough parallelism available. Capabilities of parallel execution of the code described in Adapt language is presented on rich set of examples of image processing operations. The approach presented in this paper has common image processing execution model, which is based on parallel processing of rows of input images.

The concepts of an active, self-configuring image processing chain represented by directed graph are introduces in [6] by Manfred Prantl and Axel Pinz. The proposed structure has a form of a directed graph (DAGs) whose paths define chains of image processing modules. The parameters of image processing chains are adapted automatically. The work represents an outline of integration of vision modules into a network of interconnected modules. Automatic graph adaptation is restricted to setting of parameters of image operations; thus, graph structure must be set up by a human expert. System function is demonstrated on an example of construction of image processing chain for extraction of optic disc from set of scanning laser ophthalmoscope images. Production of image processing chain represented by graph is also within the scope of this paper.

Part of work [3] introduced by Simon J. Del Fabbro concerns development of a dataflow visual programming language VPL. The VPL language allows users to de-

velop programs out of chains of image operators. These programs (image processing chains) are then executed by Java Advanced Imaging platform. The VPL language allows user to develop graphical representation of a dataflow program, which can be executed directly. The structure of visual programs is described by directed acyclic graphs (DAG). The execution is dataflow driven. Function blocks called nodes are executed (fired) when all incoming arguments become available on incoming DAG arcs. When a function is fired, it consumes incoming data and produces a single data object on the outgoing arc. The described dataflow approach is similar to that presented in this paper.

Paper [7] outlines near-term future of parallel computing, dominated by medium-grain distributed memory machines in which is each processing node represented by desktop workstation. The paper also describes design and implementation of Parallel Image Processing Toolkit (PIPT) library. The proposed library is easily extensible, and hides parallelism from the user. Message-passing model of a parallelism is designed around the Message Passing Interface (MPI) standard. Cluster based parallelism and mainly data distribution are discussed in detail. Window based operations are assigned on per pixel basis to every used processor. PIPT library uses manager/worker scheme in which a manager process reads image file, partitions it into equally sized slices, and sends the pieces to worker processors.

2 MOTIVATION FOR YET ANOTHER SCRIPT LANGUAGE

Real-time image processing applications often exploit hardware accelerated units in order to reduce the time complexity of the selected image processing operations. This approach has one important disadvantage, namely the time required for development and testing of such hardware units. The time required for design and testing of software unit performing the same functionality is much shorter.

The main motivation for design and implementation of the proposed script language is the solution of the above-mentioned problem or at least reduction of its impact. Similarly to pure software solutions, the (SLIP) language enables design of image processing applications through description of the interconnection of the building blocks representing basic image functions.

This approach enables rapid prototyping and testing of image processing operations that can be moved into the hardware accelerated units after their debugging (at the level of hardware FPGA connections equivalent to the C/C++ functions used in the script). The SLIP language was designed with the need for implementation of its interpreter for embedded processor in mind (e.g. DSP). The reason for using embedded processors is the need for control of the attached FPGA circuits, their (re)configuration, and data exchange.

Despite the fact that the SLIP language was designed mainly for image processing applications, it is also possible to use it as general purpose programming language.

3 SCRIPT LANGUAGE FOR IMAGE PROCESSING

The proposed SLIP language processing system is intended as a part of image processing applications. An application typically consists of the following components: Control application designed for communication with users, interpreter of SLIP scripts running in a PC or in embedded processor – DSP, data stores designed for sharing data and images among the system parts, and optionally an FPGA array containing selected of operation blocks. The structure of the described system is depicted in Figure 1.

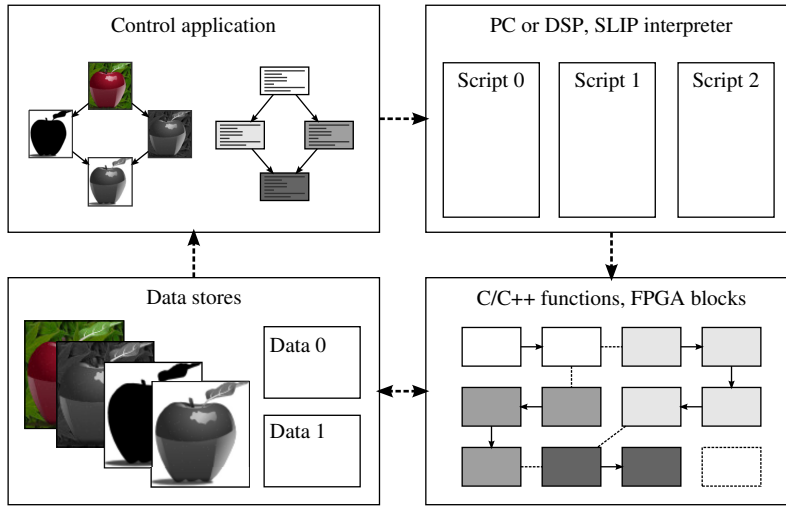

Fig. 1. Structure of image processing system

The above-mentioned control application contains graphical user interface GUI, tool for script editing and a component for manipulation with script data stores. Setting of the system enables execution of selected scripts on given data (images) through interaction with GUI of the control application. The default script execution mode of image operations uses C/C++ functions implementing the basic building blocks, which can be replaced by the blocks defined in FPGA, etc. In this system, data stores serve as communication channels between the system components (for data handling and transfers).

This paper aims at description of the SLIP language itself and its specific features for image processing. The rest of the system, such as image processing blocks, is not covered in detail.

3.1 Basic Syntax of the Language

The basic syntax (syntax describing the program flow, data types, classes, and construction of objects) of the SLIP language is derived from the well known general purpose programming language Java.

The script language is designed as a dynamically typed language enabling dynamic change of variable types at run-time. Every language element must form a part of some superior class that serves as an access point (from global point of view) to the element. The SLIP language was designed as strongly object oriented. This approach simplifies the interpretation process but the main gain from this approach is significant simplification of image processing operation graph description. All language objects are derived from some object class; even built in data types (integer numbers, floating point numbers, strings, etc.) are represented by objects. As mentioned above, the SLIP language can be used as a general programing language with the syntax subsystem that is designed for description of image processing operations.

The scripts are identified in other modules of the image processing system through the name of the first defined class (root class).

4 IMAGE PROCESSING AND DATA STRUCTURES

As mentioned above, the SLIP language was designed mainly as a tool allowing description of image processing operations composed from basic building blocks. In this section, how this task can be done and examples of how such operations are implemented will be described in detail.

The following list shows classes describing the objects designed for description of image processing operations. (Basic data types of the SLIP language are not introduced because of their irrelevance to description of image processing operations.)

Image – class of objects designed for description and manipulation with structures representing image data. Class structure design enables sharing of image data among more images and creation of images with interleaved pixel formats.

Video – class of objects designed to mediate access to open video files and video devices of the host system. Devices represented by objects derived from this class can be used as input or output nodes of image processing operation.

IONode – objects derived from this class form elements of image operation graphs. Constructors of these objects accept the following as parameters: identifier of requested operation, references on outputs of previously defined nodes, and parameters modifying selected operation.

IOStream – object of this type is constructed from set of *IONode* type objects. Object state and content are determined by graph structure described by connection of each node to previous nodes through output-input mapping. In the process of *IOStream* object construction CPU multi-threaded image operation or reconfigured FPGA image operation representation is created.

4.1 Image Representation

In vast majority of image processing applications, images are represented by two-dimensional arrays, regardless of their nature and acquisition method through which

they were obtained. This image representation method is also adopted in image processing system, which includes the proposed language. Images with *8-bit* grayscale, *32-bit* grayscale, and *24-bit BGR* (Blue Green Red) pixel formats are supported. The structures designed and used for image representation allow sharing the image data among more images, thus allowing creation of images referring to sub-rectangles or even selected channels of multi channel images. The details of the structure used for image representation are shown in Figure 2.

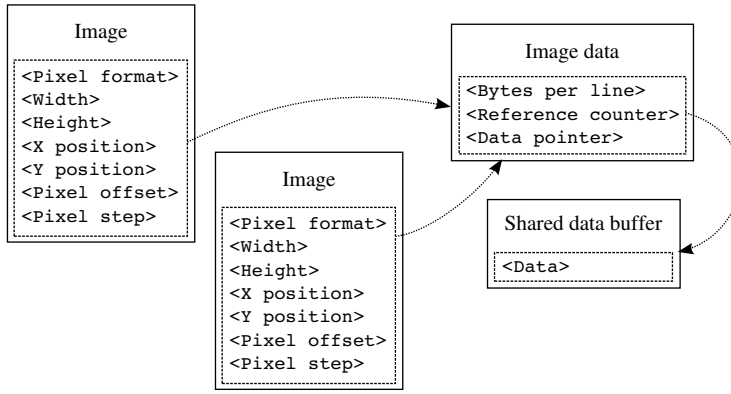

Fig. 2. The structure used for image representation

Building blocks (functions) that are used to build image processing operations work with the above-described structure of images. All of the input and output image data must be mediated through this structure. For example, the data from camera must be packed into this structure before further processing. In most cases, the mediating process is fully automatic.

4.2 Parallel Image Processing

The basic elements of which the graph of image processing operations consists are formed by objects derived from the *IONode* class. The following code illustrates the possibilities of creating such objects using their built-in constructors.

```
// - creation of new function node -
<identifier>= new IONode();
// - creation with setting of node function -
<identifier>= new IONode(<function>,[<parameters>]);
// - creation with binding of node inputs -
<identifier>= new IONode(<function>,[<node_inputs>],[<parameters>]);
// - change of node function -
<identifier>.function(<function>,[<parameters>]);
// - rebind of input nodes -
<identifier>.inputs([<nodes>]);
```

The operation performed by an *IONode* class object is determined by its function identifier. The function of a node determines the count of its input and output channels that must be connected before construction of an operation stream. The output channels of the nodes are referred to by the node name and integer index of the requested output. Examples of node construction and input-output channels building are given below.

Parallelism of image operation is achieved by parallel execution of functional nodes which serve as producer of the output data based on the data consumed from its input channels. Synchronization is done through data-flow. When no input data is consumed, the functional unit is suspended and waits for input from the data producers.

The internal representation of node operations is based on data buffering – shared data buffers represent input/output node connections. Actual implementation of messages sent from producer nodes to consumer nodes is based on interprocess communication of the target architecture.

A simple example illustrating the construction of image operation nodes and connection of these nodes through image data channel is shown below.

```
// - creation of source node (no input channels, one parameter - image) -
source= new IONode(IONode.IMAGE_SOURCE,[],[new Image("image.bmp")]);
// - color operation (one input channel - source[0], two parameters - color ↩
    operation and BGR color) -
sat_add= new IONode(IONode.COLOR_OP,[source[0]],[IONode.OP_SAT_ADD,[100,0,0]]);
```

Built-in functionality performed by image operation nodes can be subdivided into separated classes by criteria of expected input data and produced results.

Point operations – in input stream receive image data, and produce statistical information; for example maximum, minimum and average values of image pixels.

Image Arithmetic – input is represented by two streamed images, on whose pixels selected operation is performed; for example image data copy, adding, product evaluation, and masking.

Geometric operations – streamed image object and matrix defining requested image operation are taken as input. The matrix content is described by array, which is given as node parameter. The given matrix defines translation, rotation, or scaling.

Convolutions operations – similarly as in previous case, except that the given matrix describes convolution kernel instead of transformation matrix (convolution matrix can also be described by second input image stream).

Histogram operations – perform histogram based analysis on input image stream. Result of operations from this class can be either integer data stream, or image stream (image containing depicted histogram).

Differential operations – take one image as input, and perform differential operations (Hessian, Gradient, Laplacian). The result is represented by output image stream.

4.3 Hardware Implementation

The operations described in the above text can be performed not only in software, but also in specialized hardware units. For experimental purposes, Field Programmable Gate Arrays (FPGAs) present one of the most suitable solutions as it can be programmed according to the application needs and possibly also reprogrammed dynamically as the program execution requires.

The functional units that perform the required operations can be interconnected through a configurable network intended for transfer of the data from and to the execution units. The main purpose of the interconnection network and existence of several execution units is motivated by the fact that execution of multiple operations in hardware is more efficient than execution of just a single operation, mainly due to the price of data transfers from the FPGA to the CPU. Therefore, the more operations are performed in hardware simultaneously, the better for the performance.

Block diagram of the FPGA based system is shown in Figure 3. As shown, the functional units can have one or several inputs – they can perform unary, binary, or n-ary operations, mostly with single results. The main limitation of the approach is that the data is sent to the units serially and to reduce complexity of the functional units, they cannot have too large internal memory and so the operations should be performed synchronously on the data stream.

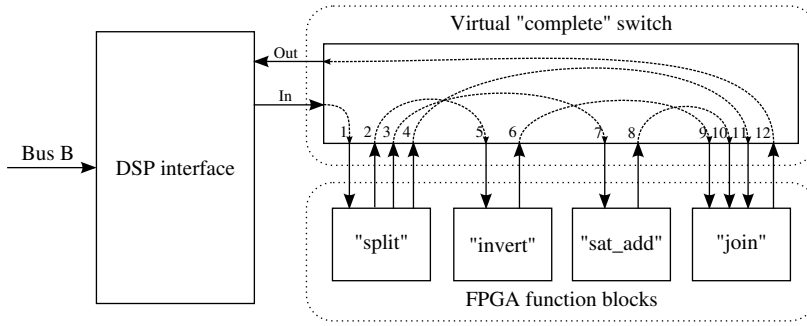

Fig. 3. Block diagram of FPGA based system

Configuration of the functional units and interconnection network is done in different time slots than the execution itself. The whole process can be subdivided into several stages:

- Detection of the expressions to be performed in FPGA (the compiler can mark the sections either implicitly or explicitly),

- preparation of the interconnection network and functional units according to the data extracted by the compiler (in runtime, based on data structures created by the compiler); this step is performed every time before execution of certain part of the script
- start of the data transfer from the CPU to the FPGA and result transfer from the FPGA into the CPU (performed typically by initiation of DMA transfers synchronized by hardware),
- passive waiting for the transfer end (during this time, CPU can execute different threads, some synchronization mechanisms are needed),
- upon end of the transfer, release of the functional unit (in many cases, this actually means no further action).

The applications can benefit from exploitation of the combined CPU and FPGA execution only in some carefully selected cases where CPU and FPGA can be used simultaneously, where the data transfers are not too expensive given the complexity of the task done in the FPGA, and also given the suitable data structures. However, in cases where the execution is suitable for FPGA and CPU combination, the applications can significantly benefit from both performance and price/performance points of view.

4.4 Examples of Image Operations

The following examples illustrate two different image processing operations. The examples consist from SLIP code and the intermediate images in individual image operation steps.

```
// - access to data store by name -
store= new Store("store_name");
// - obtain image object from store -
img_obj= store.obtain("image_name");
// - create target (blank) image object -
out_img= new Image();
// - create source node from store image object -
source= new IONode(IONode.IMAGE_SOURCE,[],[img_obj.get()]);
// - split BGR image to three separated grayscale images -
split= new IONode(IONode.SPLIT_CHANNELS,[source[0]],[]);
// - invert first channel of image pixels format -
invert= new IONode(IONode.INVERT,[split[0]],[]);
// - add constant value to grayscale image of original green channel -
sat_add= new IONode(IONode.COLOR_OP,[split[1]],[IONode.OP_SAT_ADD,[100]]);
// - create BGR image by joining of selected channels -
join= new IONode(IONode.JOIN_CHANNELS,[invert[0],sat_add[0],split[2]],[]);
// - save stream in target image -
target= new IONode(IONode.IMAGE_TARGET,[join[0]],[out_img]);
// - create operation stream
stream= target.create_stream();
// - run opertion stream -
stream.run_wait();
// - set image object in data store -
img_obj.set(out_img);
```

The first block of code describes simple image operation over image loaded from data store. The result of the depicted operation is stored into the same object of data store from which it was originally loaded.

The following code describes image processing operation introducing simple edge detector based on Sobel filter. The results of horizontal filter are combined with those of vertical filter. The input frames are loaded from video file which forms one (initial) node of image operation graph.

```
// - create target (blank) image object -
out_img= new Image();
// - as source of images use video file -
source= new IONode(IONode.VIDEO_FILE,[],["video.avi"]);
// - convert source image to grayscale 8bit format -
source_8U= new IONode(IONode.CONVERT,[source[0]],[Image.PIXEL_FORMAT_8U]);
// - detect edges by Sobel filter from left to right -
lr_sobel= new IONode(IONode.SOBEL,[source_8U[0]],[IONode.OP_LEFT_RIGHT]);
// - compute difference of resulted grayscale value from 127 -
lr_sobel_diff= new IONode(IONode.COLOR_OP,[lr_sobel[0]],[IONode.OP_DIFF,[127]]);
// - same operations for Sobel filter from top to down -
td_sobel= new IONode(IONode.SOBEL,[source_8U[0]],[IONode.OP_TOP_DOWN]);
// - compute difference of resulted grayscale value from 127 -
td_sobel_diff= new IONode(IONode.COLOR_OP,[td_sobel[0]],[IONode.OP_DIFF,[127]]);
// - sum sobel values (left to right + top to down) -
sum_sobel= new IONode(IONode.OPERATOR,[lr_sobel_diff[0],td_sobel_diff[0]],[↩
    IONode.OP_SAT_ADD]);
// - normalize resulted sobel values -
sum_norm= new IONode(IONode.NORMALIZE,[sum_sobel[0]],[1]);
// - convert values to target image format -
target_3x8U= new IONode(IONode.CONVERT,[sum_norm[0]],[Image.PIXEL_FORMAT_3x8U]);
// - store values to target image -
target= new IONode(IONode.IMAGE_TARGET,[target_3x8U[0]],[out_img]);
// - create image operation stream -
stream= target.create_stream();
// - run operation on first 100 frames -
idx= 0;
do {
   stream.run_wait();
} while(++idx < 100);
```

The image processing operations depicted in the above examples illustrate the possibilities of the proposed SLIP language, specifically easy parallelization and rapid prototyping of image processing operations.

The image operation stream (chain) constructed from a graph is a parallel operation exploiting the possibilities of today's multi-core CPU's. The function executing the image processing node task is running in its own processing thread working with its assigned CPU core.

5 DATA STORES AND COMMUNICATION BETWEEN SCRIPTS

This section discusses the possibilities of data exchange between individual parts of the image processing system. The data communication is based on data stores, the wide system structure that is accessible from every part of image processing system. Description of the SLIP language basic class designed for manipulation with data stores follows.

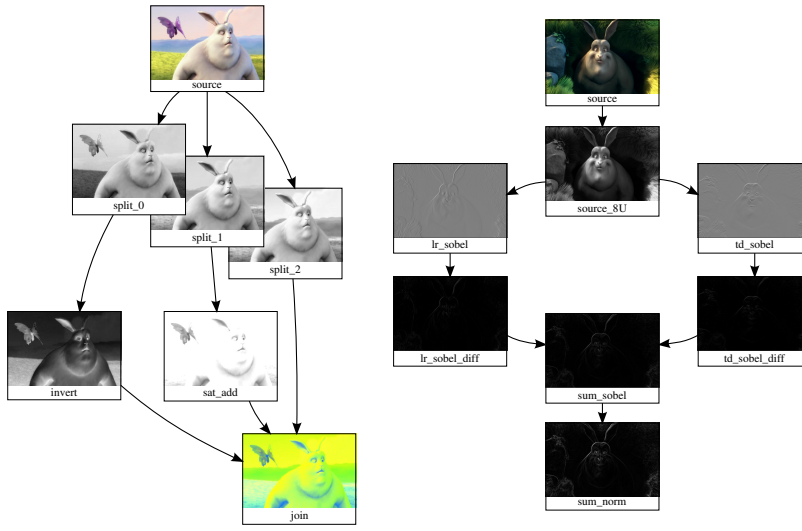

Fig. 4. Intermediate results in nodes of example image processing operations; a) Simple operation with image pixels, b) Example of image operation over data from video

Store – objects of this class mediate access to image processing system data stores. These data stores were created to enable communication among running scripts, control application, and other parts of the image processing system.

A store object is created by calling constructor of *Store* class with the name of the desired store given as a parameter. The method will either create and register new store by using system dependent interprocess communication, or get a reference to already existing registered store with the same name as that of the desired store. The newly created data store will exist as long as there is at least one reference to it.

The values shared among the scripts are accessed through a reference to data store members which mediates access to the stored data. Each of the data store objects can work with the stored variables without any restriction; for example, it can backup them or synchronize them with a database. The data stores were designed mainly for manipulation with processed images and they are implemented through shared memory of the host system. Basic syntax of data store operations follows.

```
// - creation of new or access to existing data store -
<store_id>= new Store("store name");
// - cretion of new store object or retrieve reference to existing one -
<object_id>= <store_id>.obtain("object name");
// - retrieve object from store -
<identifier>= <object_id>.get();
// - set new value to object in store -
<object_id>.set(<expression>);
```

A simple example of manipulation with image through data store is shown below. When this code is executed in two separate interpreters, the process which first accesses the shared variable loads image from file and stores it to shared variable of the data store. The second process of interpreter receives image through data stores and saves it to output file.

```
// - access to data store by name -
store= new Store("store");
// - obtain image object from store -
img_obj= store.obtain("image");
// - lock store object to avoid its change -
img_obj.lock();
// - if object type is image save it to output file -
if (type img_obj.get() == type Image) {
   img_obj.get().save_to_file("copy.jpg");
   // - unlock store object -
   img_obj.unlock();
}
// - else load image from file and save it to store -
else {
   img_obj.set(new Image("image.jpg"));
   // - unlock store object -
   img_obj.unlock();
   // - wait moment, keeping store and value active -
   System.msleep(10000);
}
```

6 CONCLUSIONS AND FUTURE WORK

This paper introduces a language that enables description and consequent execution of high-level image processing operations. Image processing operations are composed from basic building blocks. Each operation is represented as directed graph, whose edges represent interconnection of image processing blocks. Blocks (representing vertices of mentioned graph) are represented by C/C++ function or by FPGA blocks.

The image processing operation construction was shown along with description of data stores used for data handling among parts of image processing system. Examples of description of image processing operation by proposed language were shown, and feasibility of the whole approach was demonstrated.

Future work will be aimed at more expressive description of per pixel image operations, enabling description of image processing block in more specific details. This step will allow user to exploit algorithms defined apart from C/C++ or FPGA block library. Attention will also be paid to efficient implementation of the interconnection framework.

Acknowledgements

This work has been supported by the European Regional Development Fund in the "IT4Innovations" Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

REFERENCES

[1] Aho, A. V.—Sethi, R.—Ullman, J. D.: Compilers, Principles, Techniques, and Tools. In Reading MA, Addison-Wesley, ISBN: 0321486811, 2007, pp. 1–1038.

[2] DePiero, F.: Siptool: The 'Signal and Image Processing Tool' – An Engaging Learning Environment. Frontiers in Education, Annual, Vol. 3, 2001, pp. 1–5.

[3] Hawick, K.—Coddington, P.: Developing a Distributed Image Processing and Management Framework. Technical report, November 20, 2000.

[4] Gentile, A.—Sorbello, F.: Image Processing Chain for Digital Still Cameras Based on the Simpil Architecture. In Proceedings of the 2005 International Conference on Parallel Processing Workshops, IEEE Computer Society, Washington, DC, USA, ISSN: 15302016, ISBN: 0769523811, 2005, pp. 215–222.

[5] Taniguchi, R. I.—Tsuruta, N.—Kusakabe, S.—Amamiya, M.: A Massively Parallel Programming Language and Its Application to Image Processing and Computer Vision. In Proceedings of the 9 Scandinavian Conference on Image Analysis, Vol. 2, June 1995, pp. 857–866.

[6] Prantl, M.—Pinz, A.: A Concept for an Active, Self-Configuring Image Processing Graph. In proceedings of the 19th OAGM and 1st SDRV Workshop, Maribor, Vol. 81 of Schriftenreihe der OCG, 1995, pp. 249–256.

[7] Squyres, J. M.—Lumsdaine, A.—McCandless, B. C.—Stevenson, R. L.: Parallel and Distributed Algorithms for High Speed Image Processing. Technical report, Rome Laboratory, Air Force Research Laboratory, Information Directorate, Rome Research Site, 2000.

[8] Takashina, T.—Ackermann, H.: R-Based Environment for Image Processing Algorithm Design. Working paper at DSC (Distributed Statistical Computing), March, 2003.

[9] Webb, J. A.: Adapt: Global Image Processing with the Split and Merge Model. Technical report, School of Computer Science, Carnegie Mellon University, July 02, 1991.

Jiří Zuzaňák is Ph. D. student at the Department of Computer Graphics and Multimedia, Faculty of Information Technology, Brno University of Technology. His professional interests include graph grammar systems, image processing, computer vision and related research areas.

Pavel Zemčík is Associated Professor at Department of Computer Graphics, supervisor of Jiří Zuzaňák, and Vice-Dean for External Relations of Faculty of Information Technology, Brno University of Technology. His professional interests include computer graphics, image processing and computer vision, applications, and related research areas.