A Methodology for Rapid Prototyping of Real-Time Image Processing VLSI Systems

Ivan C. Kraljić, Georges M. Quénot* and Bertrand Zavidovique†

Laboratoire Système de Perception
DGA/Établissement Technique Central de l'Armement
16 bis avenue Prieur de la Côte d'Or, 94114 ARCUEIL Cedex FRANCE
Email: ik@etca.fr

Abstract

A methodology aimed at prototyping real-time image processing VLSI systems is presented. The prototyping methodology includes the emulation of the application in its target environment (i.e. in real time and on real-world scenes) and the automatic derivation of a VLSI chip-set implementing the application. Our methodology is based on a custom emulator called the Data-Flow Functional Computer (DFFC) which is dedicated to real-time image processing. Our method encompasses in a coherent environment: 1) the validation of the high level specification, 2) the simultaneous validation of an implementation of the design suitable for integration and 3) a method for integrating the validated emulator architecture as a VLSI system. The results of the successful prototyping of a defect detector algorithm are presented.

1 Introduction

Designing image processing VLSI systems can be done only intuitively using experimentation and heuristics. In this sense the full validation of the high level specification (i.e. does the specification meet the designer's idea of the application?) is a major requirement in the design of image processing applications. The validation of the specification (and *a fortiori* of the implementation) should be performed in the target environment of the application. This is not a trivial task mainly because of the tremendous amount of input data that must be processed: the computational requirements of real-time image processing range up to billions of operations per second. Conventional simulation is obviously not adapted to the validation of such algorithms (or their implementing architectures) due to its lack of power and inability to take the environmental queries into account. The increasing power of Field-Programmable Gate Arrays (FPGAs) permits the design of custom emulators (or prototyping machines) suitable for image processing (e.g. [2, 7]). Such machines are expected to validate an implementation of the design and do not address the specification validation issue. Currently they are not able to attain real-time performance for most realistic image processing applications.

We propose a methodology that allows 1) the validation of image processing specifications in their target environment, 2) the simultaneous validation of an architecture implementing the specification and suitable for integration and 3) the automatic derivation of this architecture into a VLSI chip-set. Our methodology is based on a custom emulator called the Data-Flow Functional Computer (DFFC) which is dedicated to real-time image processing [6, 8]. Its building element, the Data-Flow Processor (DFP) has been designed to implement a wide range of low- to mid-level image processing primitives. The dedication of our prototyping methodology to real-time image processing brings features that are not available on generic FPGA-based emulators in terms of efficient prototyping and integration of the VLSI chip-set.

The paper is organized as follows: section 2 presents our image processing emulator and the prototyping environment, while the derivation from emulation results methodology is presented in section 3. The results of the prototyping of a realistic image processing application (a defect detector) are presented in section 4. The advantages and limitations of this approach to rapid prototyping are discussed in section 5. Section 6 concludes the paper and discusses future work.

2 The rapid prototyping environment

2.1 Emulator architecture

The real-time image processing emulator is called the Data-Flow Functional Computer (DFFC) [8]. Data-flow is relative to the execution model (on the fly processing of digital video streams) and functional is relative to the programming model.

The emulator is a 3D $8 \times 8 \times 16$ array of Data-Flow Processors (DFPs). It can be seen as an FPGA-based hardware emulator except that its granularity is at the data-flow operator level instead of the gate level [6]. Operator data-flow graphs specified using a functional

*G.M. Quénot is now with Hyperparallel Technologies, Ecole Polytechnique, X-Pole, 91128 Palaiseau Cedex, FRANCE. Email: quenot@hyperparallel.polytechnique.fr

†B. Zavidovique is also with the Institut d'Électronique Fondamentale, Université de Paris XI, 91405 Orsay Cedex FRANCE. Email: zavido@etca.fr

language replace gate netlists. This approach has significant advantages in terms of emulation speed, ease of application specification and optimized architecture synthesis. The highly regular structure of the Data-Flow Functional Computer allows only local connections, however long distance connections are possible through DFPs configured as routing elements (an optimized place and route can yield a 30 to 50% DFP operating rate). The 1024 available DFPs allow large data-flow graphs to be implemented. Additional T800 Transputer boards can emulate high level image processing primitives.

Each DFP contains a programmable 3-stage pipelined datapath including 3 input FIFO queues and 3 output queues, an 8×8-bit multiplier, a 16-bit 2901-type ALU, a 256×9 bits RAM and a 16-bit counter. Six configurable I/O ports (bidirectional 10-bit bus interfaces) allow a DFP to access its neighbors. The datapath is configured through a programmable controller (a 64×32 bits program is provided for its description). Fifteen milliseconds are required to configure an entire DFP whatever the complexity of the implemented operator may be.

Figure 1: The Data-Flow Functional Computer

The DFFC's inputs/outputs are B/W and color cameras/monitors and low-bandwidth data-flow I/O ports. A SPARC 2 workstation is the host for the DFFC's prototyping environment (figure 1). The emulator is synchronized with the digital video pixel clock (up to 25 MHz pixel; actually, the digital video I/Os are slave to the emulator clock). The 25 MHz maximum operating frequency is not a flip-flop toggle limit but the actual speed at which every part of the DFP (datapath, state machine, memory, FIFOs) can operate whatever the complexity of the operator assigned to a DFP. Moreover, thanks to the pipeline organization of the DFPs and of the whole DFFC emulator, any data-flow graph implementing an application can also operate at the 25 MHz operating frequency whatever its complexity. Unlike FPGA-based hardware emulators, the DFFC emulator operates at the actual speed of the emulated hardware since it is optimized at the operator level. The DFFC emulator is able to emulate several millions of gates for a billion of cycles within a minute.

A custom VLSI ASIC containing 2 Data-Flow Processors has been fabricated using a 1μm CMOS technology (it contains 33K gates plus 8.5 Kbits of memory). The 2-DFP chip has a die size of 86 mm^2 in a 144-pin PGA package and its maximum operating frequency is 25 MHz. The material implementation of the DFFC consists of 8 boards; each board contains 8×8 2-DFP chips. Another board connected to the DFP array contains twelve T800 Transputers. Two 4 MByte RAM boards are currently connected to the DFFC and are mainly used to implement frame delays.

2.2 The prototyping flow

The image processing algorithm to be emulated is expressed in a Functional-Programming language (figure 2) [8]. A compiler converts this source file into an operator Data-Flow Graph which is then translated into a Data-Flow Processor Graph. The compilation/translation phases use an Operator Library containing data-flow operators implementing the language primitives. Two kinds of operators are available: single DFP operators (150 operators among which arithmetic and logic, comparators, constants, counters, 8-bit histogrammers, multiplexers...) and multiple-DFP macro-operators (50 operators among which convolvers, LUTs, 16- and 24-bit histogrammers...). This library can be extended easily with no need of intimate knowledge of the DFP's architecture.

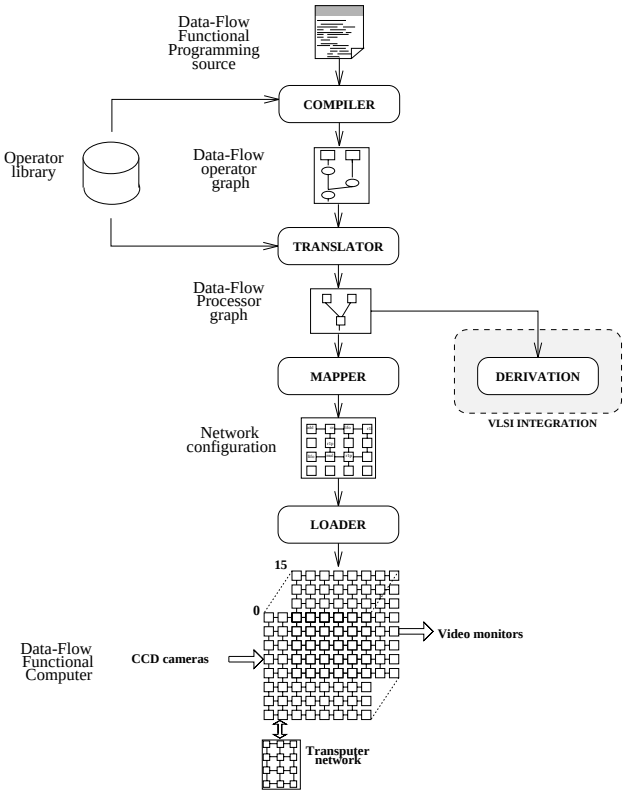


Figure 2: The prototyping flowchart

The DFP Graph is mapped into a DFFC network configuration ready to be loaded. A complete 1024 DFPs configuration is loaded in about 15 seconds (this is to be compared to the many hours needed to configure an FPGA-based emulator spent in the placement and routing of each FPGA). The algorithm is executed in real time using the CCD cameras as input flows and sending the output flows to video monitors. Debugging is done by executing the algorithm step-by-step using the low-bandwidth interfaces and the possibility to access any DFP register as well as the contents of the RAMs and FIFOs from the host (absolutely no conventional simulation is required). Several significant applications have been successfully prototyped, including colored object tracking, main direction follower and contrast enhancement [8]. Section 4 presents the prototyping of a real-time defect detector.

3 Derivation from emulation results

Once an algorithm has been validated on the Data-Flow Functional Computer emulator, a cheaper version (i.e. a VLSI chip-set implementing the algorithm) has to be generated. Rather than using high level synthesis, our idea is to use the emulator architecture (that has also been validated!) and to integrate its active resources after optimization. The integration method is called derivation from emulation results [4]. The derivation method pushes the emulation approach to its limit: the same architecture that emulated the algorithm is implemented in the VLSI system (after optimization). Derivation is rendered possible by 1) the granularity of the Data-Flow Processor which is at the operator level, and 2) the relative simplicity of the DFP which allows a deterministic knowledge of whether a DFP resource is used or not. In the following, we call *a resource* of a Data-Flow Processor any DFP datapath element (16-bit ALU, 8×8 multiplier, FIFO, register, RAM...), the DFP controller and its I/O ports. The derivation process is based on a DFP-level analysis: identification of the effectively used resources and optimization of these resources. The resulting chips architecture is pipelined FSMD (Finite-State Machine with Datapath). Derivation is similar to the retargeting of an FPGA implementation into an ASIC.

Derivation uses the 2 low level representations of an algorithm: Finite State Machine (FSM) network and RTL netlist. Low level optimizations are performed on the RTL netlist and high level transformations are performed on the FSM representation (figure 3). The validity of derivation is indicated by the average (static) resource utilization for several non-trivial algorithms (Nagao-like filter, erosion, defect detector...) which is low: 27.3% for the most important resources (from 6.7% for the multiplier up to 53.3% for the input FIFOs). As consequence a simple resource identification (i.e. removal of the unused resources) will bring dramatic size reduction of the RTL netlist. Further improvements are achieved through optimizations of this RTL netlist.

3.1 The low level optimizations

These optimizations are performed on the RTL netlist with input from the Finite-State Machine definition of each processor. They are applied to each processor and to connections between adjacent processors. The optimizations performed are: 1) a removal of the unused resources (a resource is unused if it produces no output flow or if its output flow is equal to its input flow or if it produces a constant flow); 2) an adjusting of the bitwidths of each datapath resource and of the depths of each I/O FIFO; and 3) a reduction of the programmable resources ("freezing" the controller and I/O ports to their specific function). As an example, a DFP implementing a 16-bit adder whose 8 MSBs are not used is basically reduced to a "one-stage" pipeline containing an 8-bit adder plus 3 two-word I/O FIFOs. This immediate optimization step brings huge area reduction.

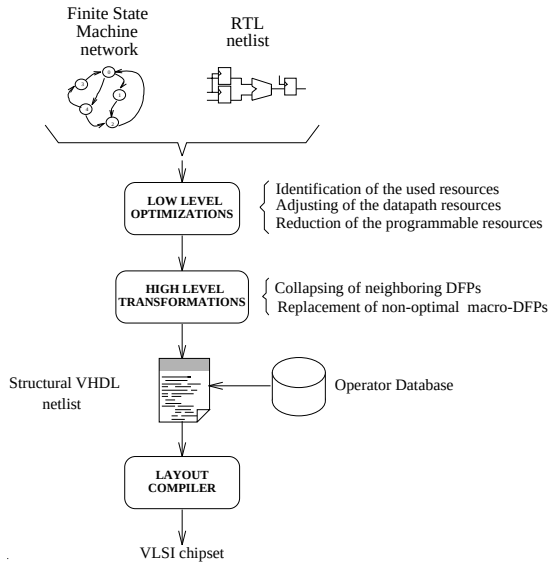


Figure 3: The derivation flowchart

3.2 The high level transformations

The low level optimizations reduce each DFP to its minimal equivalent implementation. This implementation is a data-flow operator, hence it still includes costly resources dedicated to flow management (I/O FIFOs and ports). These resources are eliminated through collapsing of neighboring DFPs into macro-DFPs according to the function they implement or to user specification. For instance, the `atan` arc tangent operator of the defect detector (figure 4) required a 1024×8 Look-Up Table (LUT) and was emulated using 9 DFPs (4 LUTs, 5 selectors) due to the DFP limitations (a DFP contains a 256×9 RAM). This operator was derived into a macro-DFP containing a 1024×8 ROM.

Output of the derivation process is a VHDL netlist at the Register-Transfer Level which is fed into a commercial layout compiler (COMPASS). Derived circuits consist of both macro and single Data-Flow Processors implemented in standard cells and datapath compiler cells. With a 1μm CMOS technology the average size of a derived DFP featuring only low level optimiza-

Operator	Initial DFP	abs	add	and	max	256-word FIFO	Pixel delay	Line delay	Histogram
Area (mm^2)	35.64	2.6	4.8	3.6	3.9	4.9	4.1	3.9	7.1
Controller (mm^2)	8.54	0.18	0.27	0.16	0.17	0.08	0.55	0.46	0.54
Datapath (mm^2)	4.85	0.76	1.30	0.94	1.38	1.09	1.25	1.00	1.94
Controller/Datapath	1.76	0.24	0.21	0.17	0.12	0.07	0.43	0.47	0.28
FIFO/Operator	0.11	0.48	0.52	0.50	0.47	0.25	0.45	0.47	0.26

Table 1: Initial and derived DFPs in a 1μm CMOS technology

tions is 4.4 mm^2 (table 1); the high level transformations decrease this number by about one half. Hence 50 derived DFPs can be safely integrated on a single chip. The use of a 0.7μm technology shall raise this density up to 100 DFPs/chip. If the derived circuit must be the smallest possible for a given large application, one can envision the use of a multichip module technology.

The derived ASICs are validated as follows: 1) each derived DFP (or collapsed macro-DFP) is validated *if it has not already been derived and validated* and 2) the whole derived ASIC is validated. Typically a few dozens input vectors are enough.

4 Prototyping a defect detector

4.1 Emulation

This section presents the results of the prototyping of an image processing application. The algorithm consists in identifying and locating defective regions in strongly patterned images (e.g. wafers) by considering edge directions. It is based on a model of human vision presented in [3]. Basically the detector identifies the regions where the edges have directions that are weakly represented in the whole image (a pattern defect is defined as a difference of the local average edges direction compared to a global measure of these directions). The data-flow graph of the defect detector is shown in figure 4. The functional programming source of the defect detector is shown in appendix A. The algorithm involves about 70 instructions/pixel, thus at a rate of 25 images/second (considering 572×768 images) the computing power required is about 800 MIPS.

The defect detector has been emulated on the Data-Flow Functional Computer in real time using 108 Data-Flow Processors (we have considered 4 directions). Table 2 shows the static resource utilization for the defect detector. Results of the emulation of the defect detector are illustrated in figure 5.

The cost of the defect detector (in terms of the number of DFPs involved) has been dramatically reduced by the programming of "application-specific" Data-Flow Processors. Indeed, the basic database DFP operators are not necessarily the cheapest implementations of a given data-flow operator. The extraction macro has thus seen its DFP count reduced from 36 to 21, and the direction macro from 25 to 21.

4.2 Derivation of the extraction macro

The direction extraction macro of the defect detector has been successfully derived using the derivation

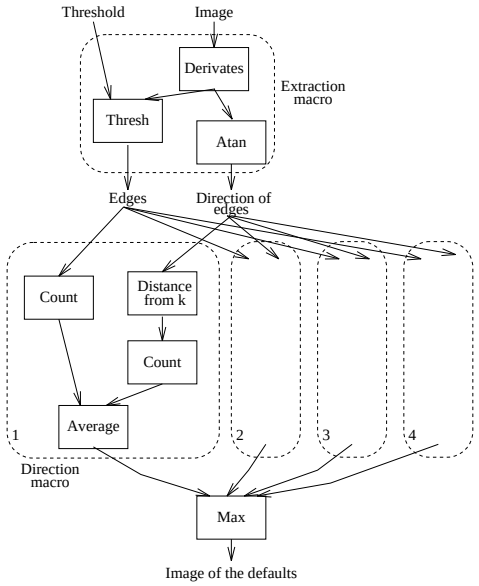


Figure 4: Data-flow graph of the defect detector (4 directions)

tool and the COMPASS gate-level and layout synthesis tools. The direction extraction chip (figure 6) contains 21 initial DFPs in 50.7 mm^2 (21K gates, core size 35.7 mm^2, 1μm CMOS ES2 technology). This chip has been derived with both low and high level optimizations, it features a gain in area of about 60% compared to the extraction chip derived with only low level optimizations. The chip has been validated by simulation using only a few dozens input vectors at a clock frequency of 12 MHz. The chip alone can process images with at most 512 pixels/line due to the on-board 512-word FIFO; but the connection of an external FIFO increases this limit.

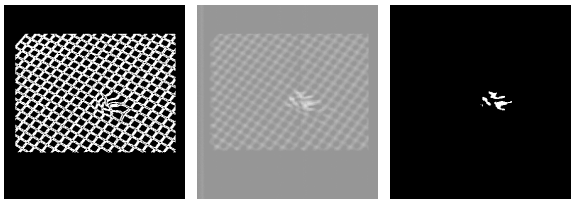
5 Advantages and limitations

Our methodology has several advantages in terms of efficient prototyping and integration of the VLSI system.

- Efficient prototyping. The availability of a prototype occurs *extremely early* in the design cycle: indeed, our methodology allows to validate a behavioral description without simulation and

Macro	I/O Ports	input FIFOs	output FIFOs	multiplier	ALU op	Data RAM	Counter	Prog RAM size
Extraction	36%	44%	27%	3%	19%	12%	12%	3%
Direction	38%	47%	30%	6%	26%	10%	30%	6%
Defect detector	38%	46%	29%	5%	25%	10%	26%	5%

Table 2: Static resource utilization for the defect detector



(a) Input (b) Distances (c) Result

Figure 5: Images produced during emulation of the defect detector

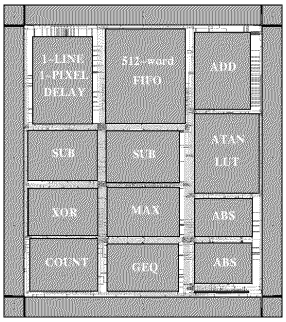


Figure 6: Floorplan of the direction extraction chip

without synthesis of a gate-level netlist (by limiting the prototype granularity to the operator level). Mere minutes are enough to obtain a working prototype from a high level specification. Both specification and an architecture suitable for integration are validated simultaneously *in real time and in the target environment of the design*. Functional validation at the actual speed of the hardware helps to find an *optimal specification of the application*: at a rate of 25 images per second, thousands of images can be processed in a few minutes thus allowing a near-exhaustive validation of the algorithm (and the discovery of bugs in the algorithm that could not be found if only a few images were processed). The ability to process many image sequences from different scenes as well as the possibility to modify parameters of the algorithm (thresholds, filters...) on the fly greatly enhances the robustness and efficiency of designs. Debugging on the emulator is to be compared to the FPGA-based rapid prototyping debugging where the designer must handle gate-level primitives (simple gates and flip-flops), whereas with our method the designer handles operator-level primitives (adders, FIFOs, multipliers...). *The observability of the prototype is total*: one can access any resource of any Data-Flow Processor.

- Fast integration. In the authors' opinion one cannot keep the integration of the VLSI system apart from the prototyping methodology. Any prototyping method is greatly weakened if the integration method is not adapted. For instance, the resynthesis of the VLSI system from the initial behavioral description does not benefit from the whole prototyping and requires much user effort (mainly in validation). Considering an FPGA-based prototype, the automatic conversion of an FPGA into a mask-programmed gate array offered by FPGA vendors [9] is interesting if the prototype is contained in a few FPGAs but totally sub-optimal if the prototype occupies more FPGAs. The integration method should be adapted to the prototyping method such as to be *a natural continuation* of the prototyping (i.e. there should be a linear flow from behavioral specification down to VLSI system). Our integration methodology is called derivation *from emulation results*: the emulator architecture that was validated during prototyping is implemented (after optimization) as the final VLSI system. This guarantees an easy and fast generation of a VLSI system implementing the algorithm with the same performance and behavior as the prototype. Derived circuits need only the smallest amount of simulation (since they are optimized versions of already validated circuits) compared to what amount would have been needed have the circuits been synthesized from a high level specification.

Such features could not be possible without a few limitations:

- Target applications. The target applications are limited to a class of low- to mid-level real-time image processing. Nonetheless, a large number of primitives fitting in this class (convolutions, filters, histograms, feature extractions, shape tracking...) can be emulated and integrated as a VLSI system. Higher level treatments can be emulated (thanks to the Transputer board) but cannot be derived. However, the functional decomposition

paradigm allows generating VLSI systems composed of derived chips and Transputers. The algorithm must be specified as a functional program or data-flow graph.

- Prototype constraints. Each data-flow node (operator) must be implementable on a single Data-Flow Processor or a macro of multiple Data-Flow Processors. Moreover, the whole application must fit into 1024 DFPs (including routing DFPs). As an example, the defect detector (see section 4) occupies 108 DFPs. Furthermore, due to the restricted structure of the DFP (8×8 multiplier, 2901-type ALU), a particular image processing primitive can be unefficiently implemented. The `atan` arc tangent operator of the defect detector has already been cited (see section 3.2). Another example is a 16×16 unsigned multiplication which requires 10 DFPs. Other primitives such as recursive functions or dynamic convolutions cannot be implemented on the emulator.

As long as the addressed systems fit into the real-time image processing domain, the limitations are very loose: data-flow graphs are well-adapted to the representation of image processing algorithms [1] and the Data-Flow Processor has been designed to implement efficiently a wide range of (low- to mid-level) image processing operators. In fact any application can be validated and integrated by our methodology if it respects the limitations.

6 Conclusion

A methodology for the rapid prototyping of real-time image processing VLSI systems has been presented. At the heart of our methodology lies a custom emulator dedicated to real-time image processing: the Data-Flow Functional Computer containing 1024 Data-Flow Processors in a 3D array. Dedication to image processing allows our emulator to operate in real time (video streams are processed on the fly) and in the target environment of the design. Furthermore, it allows the development of an integration process tightly coupled with the emulation: the derivation from emulation results methodology. The emulator architecture that has been validated during emulation is implemented as a VLSI chip-set after optimization. A realistic example of an image processing application (a defect detector) has been successfully emulated; a 21-DFP macro has been derived using our derivation tool and COMPASS' back-end tools.

Our current work consists mainly in improving the derivation process: we intend to generalize the use of datapath compiler cells in derived circuits in order to reduce their area and improve their performance [5]. Another area of research is the optimal partitioning of Data-Flow Processor graphs into derived chips. The presented defect detector application can be partitioned naturally (according to the data-flow graph hierarchy) into a derived chip implementing the extraction macro and 2 chips implementing 2 direction macros each (see figure 4). It is not proven that this natural partitioning is the most optimal one in terms of area and/or performance of the derived chip-set.

Acknowledgments

The authors would like to thank Stéphane Praud and Christophe Coutelle who have prototyped the defect detector.

References

[1] E. Allart and B. Zavidovique. Functional image processing through implementation of regular data-flow graphs. In *Twenty First Annual Asilomar Conference on Signal, Systems and Computers*, Nov 1987. Pacific Grove, CA, USA.

[2] P.M. Athanas and A.L. Abbott. Image Processing on a Custom Computing Platform. In *Proc. of the 4th International Workshop on Field-Programmable Logic and Applications*, pages 316–322. Springer Verlag, 1994.

[3] V. Brecher. New techniques for patterned wafer inspection based on a model of human preattentive vision. *SPIE Journal on Applications of Artificial Intelligence : Machine Vision and Robotics*, 1708:452–459, 1992.

[4] I.C. Kraljić, G.M. Quénot, and B. Zavidovique. High-Level Synthesis by Systematic Derivation of Vision Automata from Emulation Results. In *IFIP Workshop on Logic and Architecture Synthesis*, pages 183–187, December 1994. Grenoble, France.

[5] R. Leveugle and C Safinia. Generation of optimized datapaths: bit-slice versus standard cells. In G. Saucier and J. Trilhe, editors, *Synthesis for Control Dominated Circuits (IFIP Transactions A-22)*, pages 153–166. Elsevier, 1993.

[6] G.M. Quénot, I.C. Kraljić, J. Sérot, and B. Zavidovique. A Reconfigurable Compute Engine for Real-Time Vision Automata Prototyping. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 91–100, April 1994. Napa, CA, USA.

[7] A. Saha and R. Krishnamurthy. Some Design Issues in Multi-chip FPGA Implementation of DSP Algorithms. In *Proc. of the 5th International Workshop on Rapid System Prototyping*, pages 131–140. IEEE Computer Society Press, 1994.

[8] J. Sérot, G. Quénot, and B. Zavidovique. Functional programming on a dataflow architecture: applications in real-time image processing. *Machine and Vision Applications*, 7(1):44–56, 1993.

[9] Xilinx. *HardWire Data Book*, 1994.

A Defect detector functional programming source

```
// ————————————————————
// Defect detector (4 directions)
// ————————————————————

//****************
// Direction macro
//****************
BEGIN dirk [
      INPUT direction:FRAME(PIXEL);
```

```
      INPUT edge:FRAME(PIXEL);
      INPUT counter:PIXEL;
      OUTPUT contribk:FRAME(PIXEL);
      ]

  DEF I = MUX(0). [RAM . direction, edge];
  DEF CH = 1.ADD . [ADD1P2P . [I,I,I], I];

  DEF mem512 = FIFO.FIFO.CH;    DEF CV = ADD1L2L .
[mem512, FIFO.mem512, CH];
  DEF IF = MUX(0) . [CV, edge];
  DEF LST = R1P . [IF, IF];
  DEF LESS = AND . [LEQ . [LST, IF] , XOR(255). EQ . [LST,
IF]];
  DEF RES0 = COUNT . LESS;
  DEF ol = 1 . RES0;

  DEF INV = RAM . counter;
  DEF MUL16 = MUL . [INV, ol];
  DEF MULH = 2 . MUL16;
  DEF MULL = 1 . MUL16;
  DEF AVERAGE = ADD . [MULL, MULH];

  DEF INV2 = RAM . 1.AVERAGE;
  DEF contribk = MUL . [INV2, IF];
END

//******************
// Macro - extraction
//******************
BEGIN extraction [
      VIDEO INPUT image:FRAME(PIXEL);
      VIDEO INPUT threshold:PIXEL;
      VIDEO OUTPUT direction:FRAME(PIXEL);
      VIDEO OUTPUT edge:FRAME(PIXEL);
      VIDEO OUTPUT counter:PIXEL;
      ]

  // 1-line and 1-pixel delay
  DEF Delay1l1p = R1L1P . [image, FIFO.FIFO.image, image];
  DEF Delay1p = 1 . ret1l1p;
  DEF Delay1l = 2 . ret1l1p;
  DEF dx = SUB2 . [image, Delay1p];
  DEF dy = SUB2 . [image, Delay1l];
  DEF angle = XORSGN . [dx,dy];
  DEF adx = ABS4 . dx;
  DEF ady = ABS4 . dy;

  // Atan arc tangent operator
  DEF dxdy = CONCAT . [adx, ady];
  DEF lut0 = RAM . dxdy;
  DEF lut1 = RAM . dxdy;
  DEF lut2 = RAM . dxdy;
  DEF lut3 = RAM . dxdy;
  DEF lsbDX = AND(1).adx;
  DEF mux0 = MUX . [lut0, lut1, lsbDX];
  DEF mux1 = MUX . [lut2, lut3, lsbDX];
  DEF atan = MUX . [mux1, mux0, AND(1).ady];

  DEF direction = ADD254 . [atan, angle];
  DEF edge1 = THR . [MAX.[adx, ady], threshold];
  DEF counter = COUNT . edge1;
  DEF edge = edge1;
END

//**********************
// Main - Defect detector
//**********************
MAIN [
      VIDEO INPUT picture:FRAME(PIXEL);
      VIDEO INPUT threshold:PIXEL;
      VIDEO OUTPUT defect:FRAME(PIXEL);
      ]

  DEF extract = extraction . [picture,threshold];
  DEF direction = 1.extract;
  DEF edge = 2.extract;
  DEF counter = 3.extract;

  DEF dir0 = dirk(k=0) . [direction,edge,counter];
  DEF dir1 = dirk(k=1) . [direction,edge,counter];
  DEF dir2 = dirk(k=2) . [direction,edge,counter];
  DEF dir3 = dirk(k=3) . [direction,edge,counter];

  DEF max0 = MAX . [dir0,dir1];
  DEF max1 = MAX . [dir2,dir3];
  DEF defect = MAX . [max0,max1];
END
```