# SPECIFICATION AND MODELING OF A CANNY EDGE DETECTOR FOR EMBEDDED SYSTEMS DESIGN

ECPS 203

BY
TINTU JOSE

12th December 2017

## Abstract

This project is based on the methods and techniques for specification, synthesis and performance modelling at system level. Here we do modelling of embedded systems that are targeted mainly for SoC implementations using SystemC. Using this SystemC language we specify, simulate, analyze, model and design systems based on Canny Edge Detector example.

# INTRODUCTION

## Embedded system modeling and design concepts

Embedded computer systems are ubiquitous, integrated into many devices we interact with on a daily basis. Driven by ever increasing application demands and technological advances that allow us to put complete multi-processor systems on a chip (MPSoCs), system complexities are growing exponentially. Together with tight constraints and market pressures, this makes the system design process a tremendous challenge and well-defined design methods and design automation techniques crucial to its success [1].

## The IEEE SystemC language

SystemC is an ANSI standard which is a hybrid part of both hardware and software. It belongs to C++ class library for systems and hardware design. The main purpose of this SystemC language is to provide a C++ standard for designers and architect who can address complex systems between hardware and software.

SystemC class library makes use of TLM library so that a SystemC implementation can be developed with reference to this standard[2]. SystemC is a system design and modeling language. This language evolved to meet a system designer's requirements for designing and integrating today's complex electronic systems very quickly while assuring that the final system will meet performance expectations[3].

# CASE STUDY OF CANNY EDGE DETECTOR FOR REAL TIME VIDEO

## Structure of the Canny Edge Detection Algorithm

The Canny Edge Detector algorithm takes an input image and calculates an output image that shows only the edges of the objects. It is useful to extract the structural information from vision objects and dramatically reduce the amount of data to be processed.

Process of Canny edge detection algorithm is divided into 5 steps[4]:

1. Apply Gaussian filter to smooth the image in order to remove noise.
2. Take the dx and dy the first derivatives.
3. Compute the magnitude of the gradient
4. Perform non-maximal suppression to get rid of spurious response to edge detection
5. Track edge by hysteresis- Finalize the detection of edges by suppressing all the other edges that are weak and are not connected to strong edges.

The overall project goal is to design a suitable embedded system model of this application and describe it in the IEEE SystemC System-Level Description Language (SLDL). The created embedded specification model will then be simulated for functional and timing validation.

**Setup, introduction to application example Canny Edge Detector**

The first step of this project started with setting up the Linux environment and softwares such as XQuartz for the graphical tools used for image processing. In this project, we made use of C reference implementation of the Canny Edge Detector algorithm as the starting point for our embedded design model and the golf cart image as an initial input image. In this step we examined and understood the main functions that are used for application and created a corresponding functional call tree of the functions.

**fig 1(a)** Golfcart Image[5]

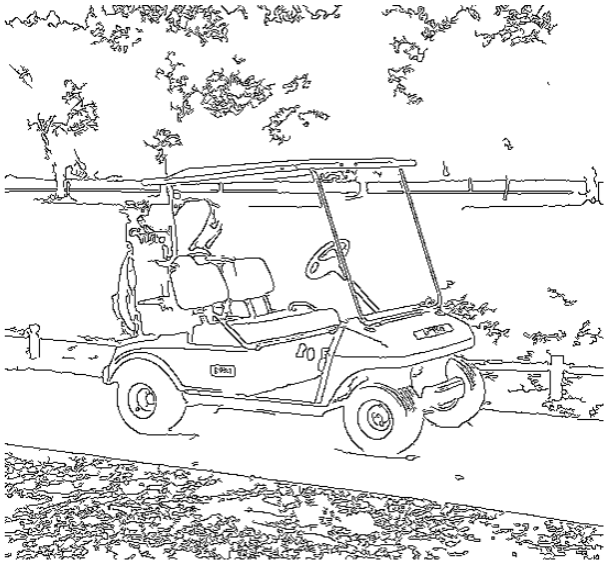

**fig1(b)** Output of Canny Edge algorithm[5]

**Clean C++ model of the Canny Edge Decoder with static memory allocation**

In second step, we refine the Canny Edge Detector application by converting C code to a C++ program as we utilize the IEEE SystemC language for simulation and validation in later steps and SystemC is based on the C++ language. In this step the Makefile is also provided for building the application model. Thus it will simplify the compilation and testing iterations. Here we have also made the configuration

parameters such as rows, cols, sigma, tlow and thigh values as constants for a future System-on-Chip (SoC) implementation.

We have also removed the dynamic memory allocation such as malloc(), calloc() and free() as it is not feasible in hardware implementation, because the desired SoC cannot instantiate a new memory chip at runtime[5] and it may lead to memory fragmentation and leakage. Thus we make use of static arrays with fixed sizes of rows and columns at compile time.

**From single image to video stream processing**

In third step, we continue with the modeling of our application example, the Canny Edge Detector. Here we convert the application of single image processing to video stream handling. More specifically, we use the video stream in a test bench of a SystemC model, thus we extract a sample set of frames and convert those to grey-scale images that our Canny application can process. For this video stream, we make use of the video captured by flying drone over the Engineering Hall building at UCI. In our design and development in SystemC, we need only few frames for processing in our test bench so we extracted only 30 frames from the stream, representing 1 second of real time video at 30 frames per second (FPS) [6] and store it in the video folder. We made use of ffmpeg software package in our Linux server for the video processing. Since canny application can only read pgm input images we converted all png frames to pgm using pngtopnm and ppmtopgm linux tools. We also need to change the image size to 2704 * 1520 pixels.

Due to high resolution of image that is 2704*1520 pixels, it leads to higher memory usage of our application, so we set the stack size of a process in Linux environment in order to avoid segmentation fault as the stack size of memory is assigned to a limit and now as the program image resolution have increased which

is beyond the stack limit allotted then it may try to access the memory beyond the limit leading to segmentation fault. As I am using **tcsh** shell, I adjusted my root thread stack size using following command **limit stacksize 128 megabytes**.

**fig 2(a) Engineering Hall image frame[6]**

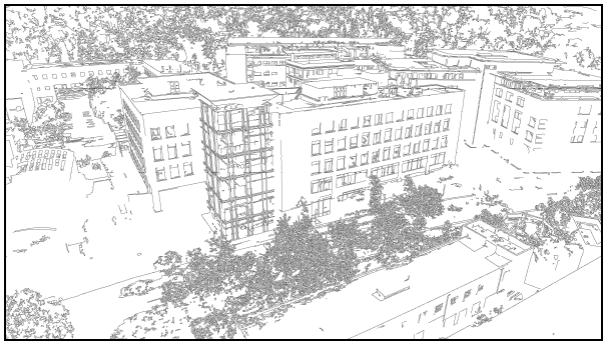

**fig 2(b) Edge image frame[6]**

**Test bench model of the Canny Edge Decoder in SystemC**

In the fourth step, we continue modeling of our Canny Edge Detector application as a proper system level specification model that is using SystemC so that we can use them to design our embedded system target implementation. In this we have 3 main modules Stimulus, Platform and Monitor. The Platform module in turn contain a Data In, DUT, Data Out modules. For communication with the modules we instantiated with fifo-type channels from SystemC standard library[7]. We use sc_fifo<IMAGE> where template parameter we use IMAGE as type of data since IMAGE is an array and C++ does not provide an operator for array assignment. To simplify this technicality, the class IMAGE is used which is provided by the instructor. In this top level structural hierarchy, a total of four channel instances are used, two at the test bench level (Top module) and the rest two within the Platform module. Moreover, the Top module should instantiate the Stimulus, Platform and Monitor modules in parallel. The Stimulus module read the input image from file system and pass it to the Platform module via channel q1 mentioned in fig 3(a). In the Platform module, the Data In module gets the image in an endless loop and pass

it unmodified to DUT via sc_fifo channel q1. The DUT module contain the entire Canny algorithm which process the image into edge image and then send it to Data Out module via sc_fifo channel q2. The Data Out module gets the edge image in an endless loop, and pass it on. From Platform module, the edge image is passed to Monitor module via sc_fifo channel q2.

Here we make use of Data In and Data Out modules as they allow our test bench to remain unmodified even when later in the design flow the communication to the DUT is implemented via detailed bus protocols.

For SystemC model due to larger images we have considered the stack size issue, one in the root thread here I have increased the stack size by providing the command **limit stacksize 128 megabytes** and the second for every SC_THREAD also we have increased the stack size by providing the statement **set_stack_size (128*1024*1024);**.

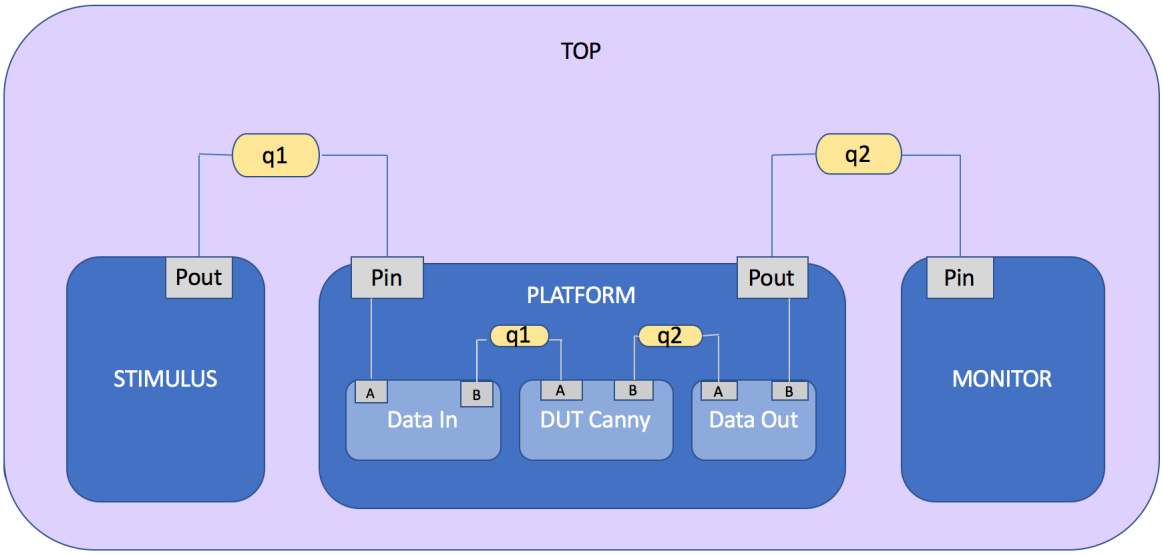


**fig 3(a) Top level structure.**

## Structural refinement of the design-under-test module and algorithm profiling

In the fifth step, we refine the Canny Edge detection algorithm model further with a suitable hierarchy inside the design-under-test (DUT) module. The original canny function consists of a sequence of function calls namely gaussian_smooth, derivative_x_y, magnitude_x_y, non_max_supp, and apply_hysteresis. While in the previous model all these are local methods in the DUT, we encapsulated them into separate modules by themselves. In this each module have its own SC_THREAD. For communication, the encapsulated modules are connected by sc_fifo channels. Here all channel instances have buffer size 1 element.

Further the Gaussian_smooth function consists of several tasks we wrapped that into three child modules, namely make_kernel, blur_x and blur_y which can be viewed in the fig 4(a). Similarly, these modules are connected by sc_fifo channels.

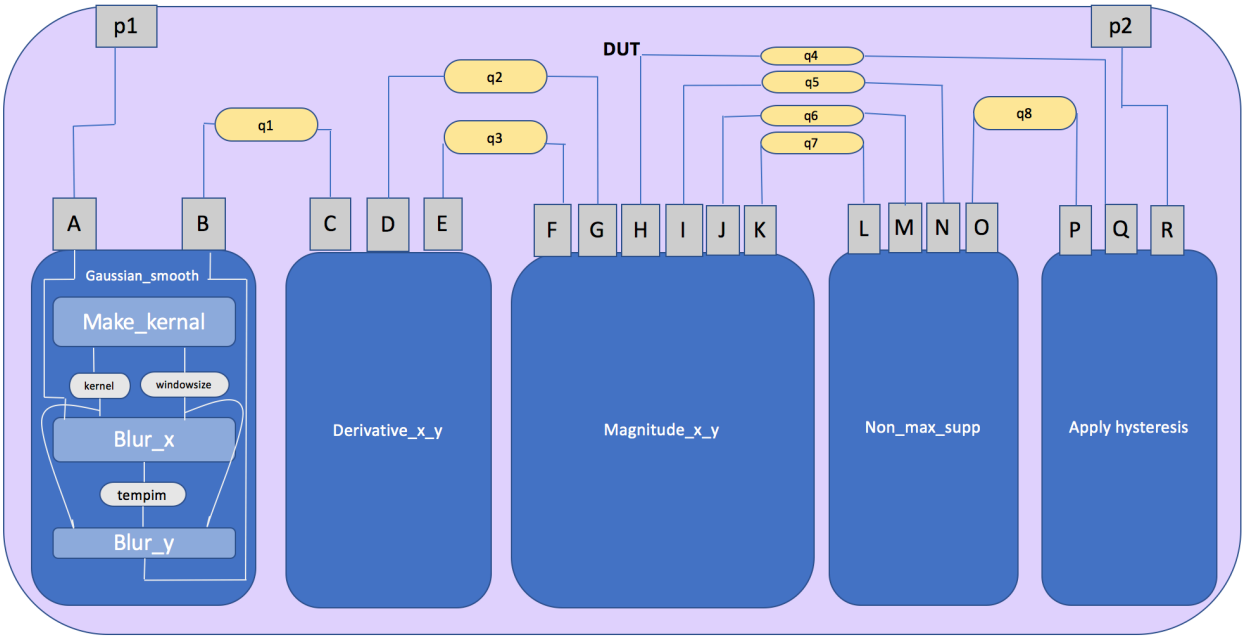


**fig 4(a) DUT level structure**

Now in this step we have done the initial performance profiling analysis of our Canny Edge Detection model, in order to identify the functions with the highest computational complexity. For SystemC model, we used profiling tools provided by the GNU community, namely **gprof**. We first instrument the model by supplying the –pg option to the GNU compiler g++. After compiling and running the executable file, this produces gmon.out file which provide the profiling statistics that can be analyzed by following command **gprof Canny.** In this profiling report we took flat profile for the computational complexity. The result of profiling is provided in Table-1.

| | | |
|---|---|---|
| Gaussian_Smooth | | 42.64% |
| \|------ Gaussian_Kernel | 0% | |
| \|------ BlurX | 22.73% | |
| \------ BlurY | 19.91% | |
| Derivative_X_Y | | 6.12% |
| Magnitude_X_Y | | 16.09% |
| Non_Max_Supp | | 25.16% |
| Apply_Hysteresis | | 9.80% |
| | | 100% |

**Table 1: Profiled complexity comparison[7]**

From the table 1 we can see that the Gaussian _smooth has the most computational complexity than other functions.

**Performance measurement of the Canny Edge Detector on prototyping board**

In this sixth step, we make use of the application source code from step 3. In order to get the absolute measurements on the target platform, we used application C++ model and measured its run time on Raspberry Pi prototyping board which will give exact measurements. These measurements will be used as an absolute reference point for performance estimation of the actual embedded system in the final

implementation. In this step, we instrumented the code with timing measurement instructions by starting the timer before the function call and stopped the timer after function call and calculating the time difference. From this we get the average real-time delays of all functions in Canny algorithm which is separate modules in SystemC model as done in the previous step5.

| | | |
|---|---|---|
| Gaussian_Smooth | | 3.53 s |
| Gaussian_Kernel | 0.00 s | |
| BlurX | 1.71 s | |
| BlurY | 1.82 s | |
| Derivative_X_Y | | 0.48 s |
| Magnitude_X_Y | | 1.03 s |
| Non_Max_Supp | | 0.83 s |
| Apply_Hysteresis | | 0.67 s |
| | | ========== |
| TOTAL | | 6.54 seconds |
| | | ========== |

**Table 2: Measured delays in Raspberry pi 3 [9].**

We can see that the measured delays obtained here varies from step 5 because in previous step we run the model in Linux servers which provides only relative values because of many users using the server at the same time the delay occurs but here we ran the code in Raspberry Pi which provides absolute results as there is no delay since users are limited to one.

From the result of the all function time delay we see that the performance is too low for real time application. So we need to improve the performance of the model in the upcoming steps.

**Pipelined and parallel design-under-test module of the Canny Edge Decoder**

In this seventh step, we refined the untimed previous SystemC model into one with estimated delays where the simulation allows us to observe the improved performance due to pipelining and parallelization. In this step we start with instrumenting the model with logging of simulation time and frame delay this is done by calculating the start time processing in the stimulus module for each frame and end the time in monitor module and store this stimulation time and delay of each frame as part 1. For the communication of time channel we make use of sc_fifo<sc_time> type. Further we need to back annotate the measured time delay which we got from step6 (refer Table 2) into our SystemC model by inserting the wait for time statements into the main method of each DUT component and store the current changed stimulation and delay time as part 2. Then we further pipeline the DUT components in order to improve the performance of the model which could be taken as part 3.

Now we do the parallelizing part which improves the throughput by slicing the Blur_x and Blur_y which we have found that these are the bottleneck modules in our model (as mentioned in table 2) into 4 threads. Here we parallelize the operations in the rows and columns where each thread operate on a one-quarter slice of the image. Technically we can operate on every row or column parallel but here we limit it to 4 parallel slices. This might speedup the process into 4 times, this can be considered as part 4.

Now after comparing the all 4 parts for stimulation time and frame delay time there is some improvement in the performance of the model which is shown in Table3.

We can also see that the simulator run time is the same as before because the code is running in the server and the total simulation time that we have calculated

are the frame delays and the total simulated time from monitor to stimulus. But the simulator run time is the time taken for the entire code to compile and execute in the server.

| Model | Frame Delay | Total simulation time |
|---|---|---|
| part 1 | 0 ms | 0 ms |
| part 2 | 23030 ms | 59320 ms |
| part 3 | 23030 ms | 59320 ms |
| part 4 | 16950 ms | 33307 ms |

**Table 3: Timing results observed in step 7.**

**Throughput optimization of the Canny Edge Decoder**

In the final step we take the pipelined and parallelized model from seventh step and this is optimized so that these pipelined stages are balanced and improved the throughput (Frames per second) of the design. We further optimized the model so that the execution time of theses stages are reduced.

In this step first we calculated the delay between two images in the monitor module and we calculated the frames per second coming out of the video. Taken this stage as part 1. We further used with compiler optimizations of step 7.

In 6$^{th}$ step we measured the timing of major canny functions on Raspberry Pi. The GNU compiler provides us many optimization options such as –O2, -O3, -mfloat-abi=hard, -fmpu=neon-fp-armv8 and –mneon-for-64bits [10]. In this we made use of –O2, after compiling with –O2 optimization the result obtained is back annotated with the part 1 which is then taken as part 2. We can see from part 2 that after compiler optimization the throughput is improved with higher FPS rate.

Now further we tried improve the throughput of the application by replacing the floating point arithmetic with fixed point calculations. We do this process for NMS module since this is the bottleneck in our pipeline model. We can also do this procedure on other components also but since nms is bottleneck of our pipeline we are trying on it and this is taken as part 3 of our process in this step. The table below shows the various simulation time, frame delay and FPS for the 3 parts which resulted in improved throughput of our application.

| | TOTAL_SIMULATION_TIME | FRAME_DELAY | FPS |
|---|---|---|---|
| part-1 | 33 secs | 1.03 secs | 0.97 |
| part-2 | 7.7 secs | 0.28 secs | 4.19 |
| part-3 | 7.7 secs | 0.28 secs | 4.19 |

**Table 4: Timing results observed in final step.**

## CONCLUSION

In this work we mainly focused for optimizing the throughput of canny edge detector application in SystemC for SoC implementation. For that we started with source code of canny edge detector algorithm which was in C language. Then we changed that to C++ language since SystemC is mainly based on C++ language. We also removed and replaced dynamic memory allocations to fixed memory sizes as dynamic memory allocation is not allowed in SystemC. We have extracted 30 of video frames suitable for use in test bench. Further we converted C++ model to SystemC model, we created each main functions of canny as modules and communicated using channels. We did profiling inorder to get the relative complexity of canny functions which we found that Gaussian_smooth as the bottleneck. We also ran the C++ model on Raspberry Pi to get absolute timing

measurements in order to instrument the source code with real time measurements. Further using SystemC model we tried to improve the throughput of the model by pipelining and parallelizing and also by compiling using compiler optimizations.

As we have done this project for only 30 frames for test bench but we can do this for all frames from the video which can be implemented in System on Chip (SoC) and can be presented to the end user as a real-time video with further improvements mentioned in future work.

**Future work**

Further we can improve the throughput of our Canny Edge Detector application till it reach the time goal of 30 FPS or more than that. We can do this by reducing the image size by decreasing the resolution as much as possible or simply accept a lower FPS rate for the end user. We can also include faster processor boards such as FPGA or DSP for complex computations which will increase the throughput significantly with the consideration of budget allowed by the end user.

## REFERENCES


[1] http://users.ece.utexas.edu/~gerstl/ee382v_f08/syllabus.html

[2] http://ieeexplore.ieee.org/document/6134619/

[3] D. Black, J. Donovan, B. Bunton, A. Keist:" SystemC: From the Ground Up, Second Edition", Springer, 2010. ISBN 978-0-387-69958-5.

[4] https://en.wikipedia.org/wiki/Canny_edge_detector

[5] https://eee.uci.edu/17f/16905/Assignment1.pdf

[6] https://eee.uci.edu/17f/16905/Assignment4.pdf

[7] https://eee.uci.edu/17f/16905/Assignment5.pdf

[8] https://eee.uci.edu/17f/16905/Lecture16.pdf

[9] https://eee.uci.edu/17f/16905/Lecture17.pdf

[10] https://gist.github.com/fm4dd/c663217935dc17f0fc73c9c81b0aa845