**Part C.1**

We implemented 1000 time units before the first start signal, and 10 time units between each image in the *Stimulus*. The total simulation time was calculated in the *Monitor*. The output of the total simulation time for each image was obtained as the following: 1000, 1010, 1020, and 1030. This indicates that the delays were purely due to the fixed delays implemented as a part of the Part C.1. requirements, and there was zero delays in the actual processing time of the images.

**Part C.2**

Relevant block diagram is attached in the reference page at the end of this report. [Reference 1]

**Part C.3**

Our model can run in bounded memory provided, each behavior runs at similar rate. If not, then, it cannot run in bounded memory. For example, *Stimulus* runs at a very fast rate but *Susan* runs very slowly. As time progresses, the size of queue grows larger and larger without an upper bound. The model can execute with queue of an image size. Limiting the queue size affects the scheduling by bring an order (corresponding to data flow) to execution. Execution order is as the following: Stimulus → read\_image → susan → write\_image → monitor.

Even though *read\_image,* *susan*, *write\_image* are parallel behaviors, they complete sequentially because of the data dependencies. *Susan* and *write\_image* will start executing but will wait on queue.receive for data from its preceding behavior. Also *read\_image* cannot execute again until *susan* gets (consumes the data in queue) once. Having queue size greater than image size, decouples the behavior from its subsequent behaviors. For example, *Stimulus* can send images continuously to *read\_image* until queue becomes completely filled. As long as all the behaviors are working as they are meant to be, and queue size is at least 1 image size, the model is deadlock free and it is deterministic, as there is no shared resource.

Relevant block diagram is attached in the reference page at the end of this report. [Reference 2]

**Part D.2**

We chose to parallelize *edge\_draw* behavior. In our implementation, we are instantiating 2 instances of *edge\_draw* behavior which processes half image in parallel. *Edge\_draw* requires 2 inputs – mid and input\_image. We created separate behavior for splitting the mid and image into 2 sections which can be sent each of the *edge\_draw* instance. Within *edge\_draw*, processing each image element modifies few image elements before and after the specific element in consideration. We split the image into 2 parts with overlapping sections, the image is 7220 (IMG\_SIZE) elements long, 1 part is image [0: (IMG\_SIZE/2 + offset)] and 2nd part is image [(IMG\_SIZE/2 – offset) : IMG\_SIZE-1], where offset is 133 (decided based on input and dependencies. The processed outputs from each *edge\_draw* instance is then merged together using another newly created behavior.

# Add behavior hierarchy sir\_tree -blt command

# graphical representation of model (screenshot from specC viewer)

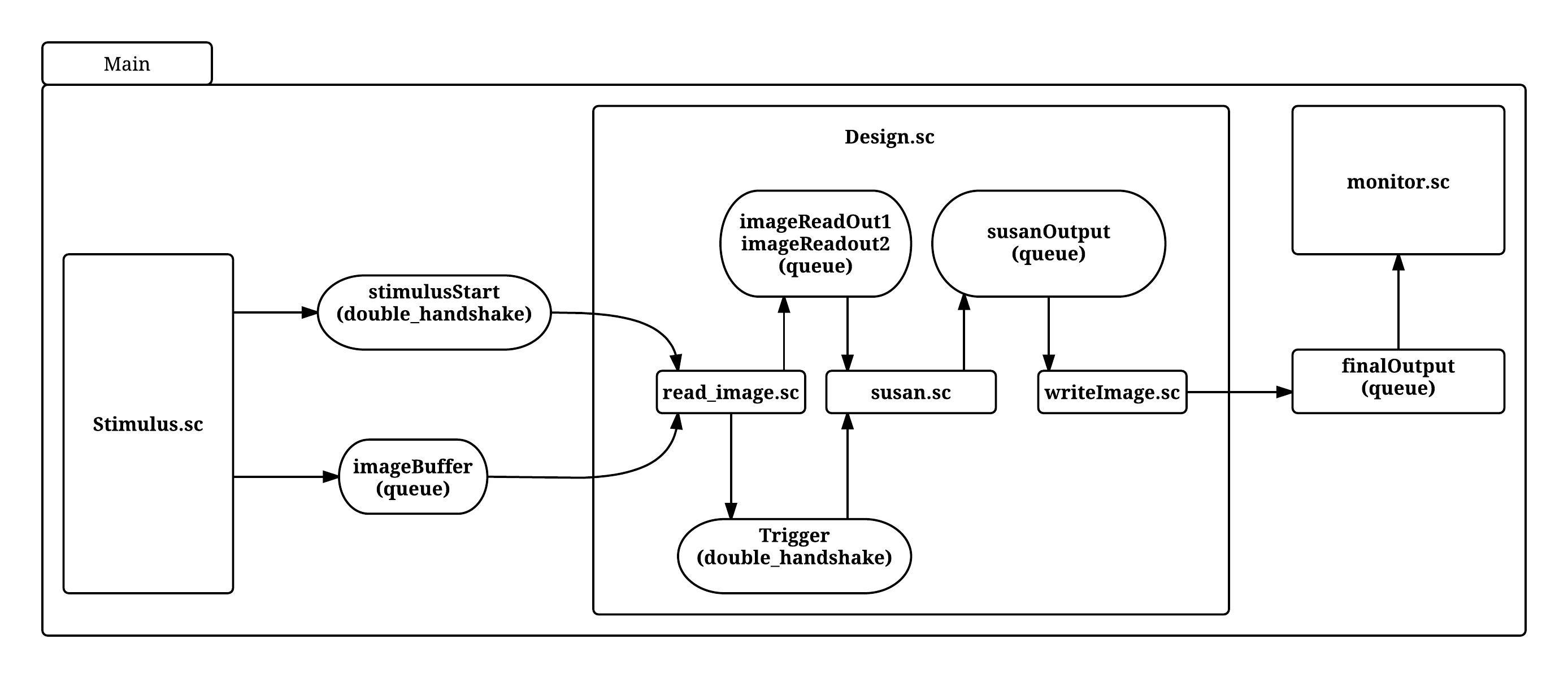
This model implemented as KPN shows the task level parallelism but the parallelism within a task is not explicitly highlighted. Potentially, there is parallelism to be exploited in *susan\_thin* and *susan\_Edge* behavior which is unknown until implemented parallel.

Implementing it using different MOC (like Data flow graph with finer granularity) would show the parallelism within task more explicitly. Yes, this model can be modeled using SDF. For SDF implementation, we will have to somehow initialize the inputs for all the parallel behaviors so that they don't wait on others for their output during the initial run.

#could a tool be developed to recognize if a model is KPN or SDF ?  
 - if feedback , KPN fails (might be wrong ?)  
 - if can statically schedule it is an SDF model ??

#Adv and Disadv for KPN vs SDF

**[Reference 1] Part C.2**



**[Reference 2] Part C.3**

