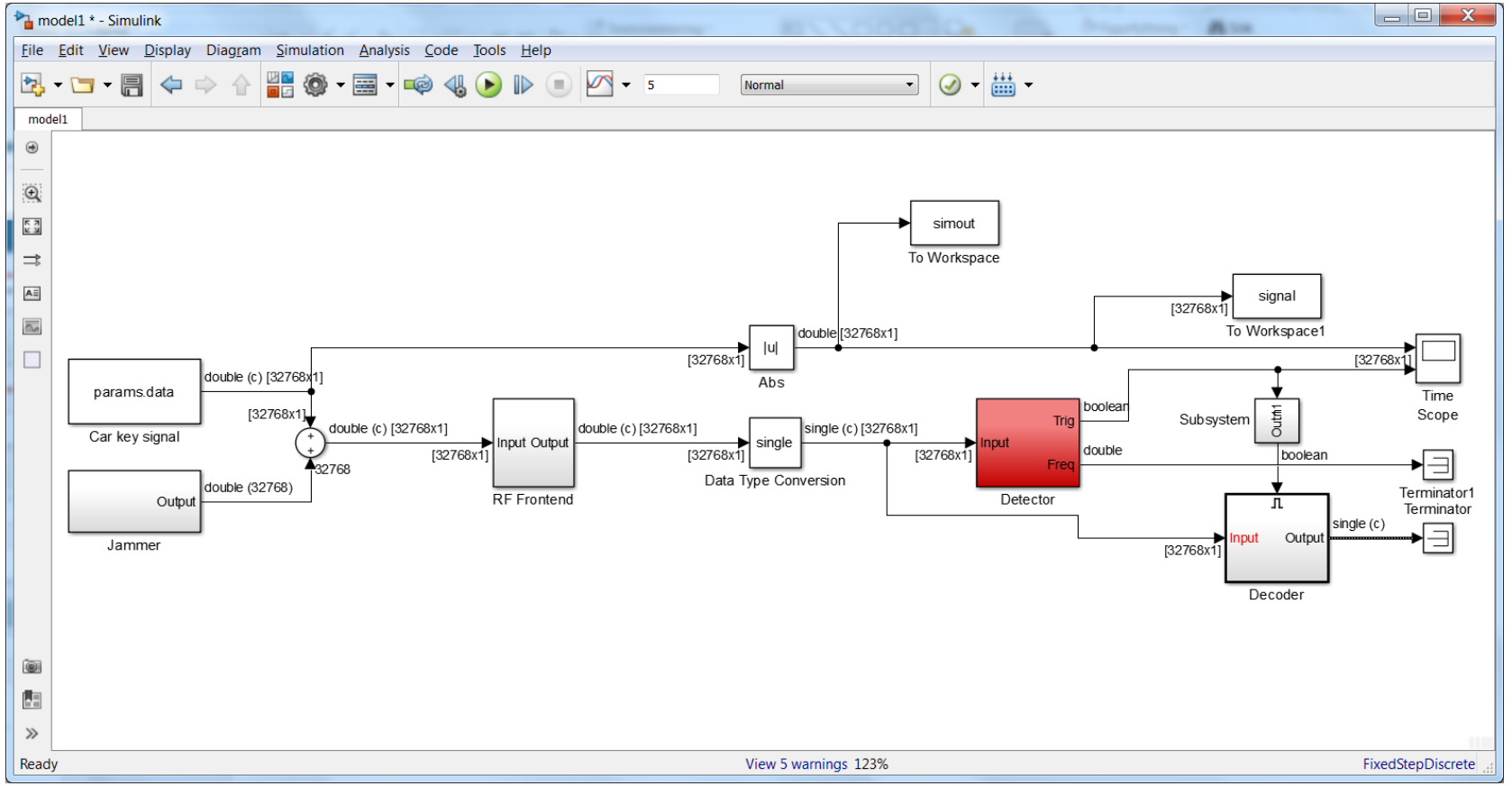
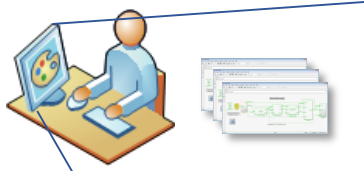
Accelerate the Design and Prototyping of Signal Processing Algorithms

Daniel Aronsson

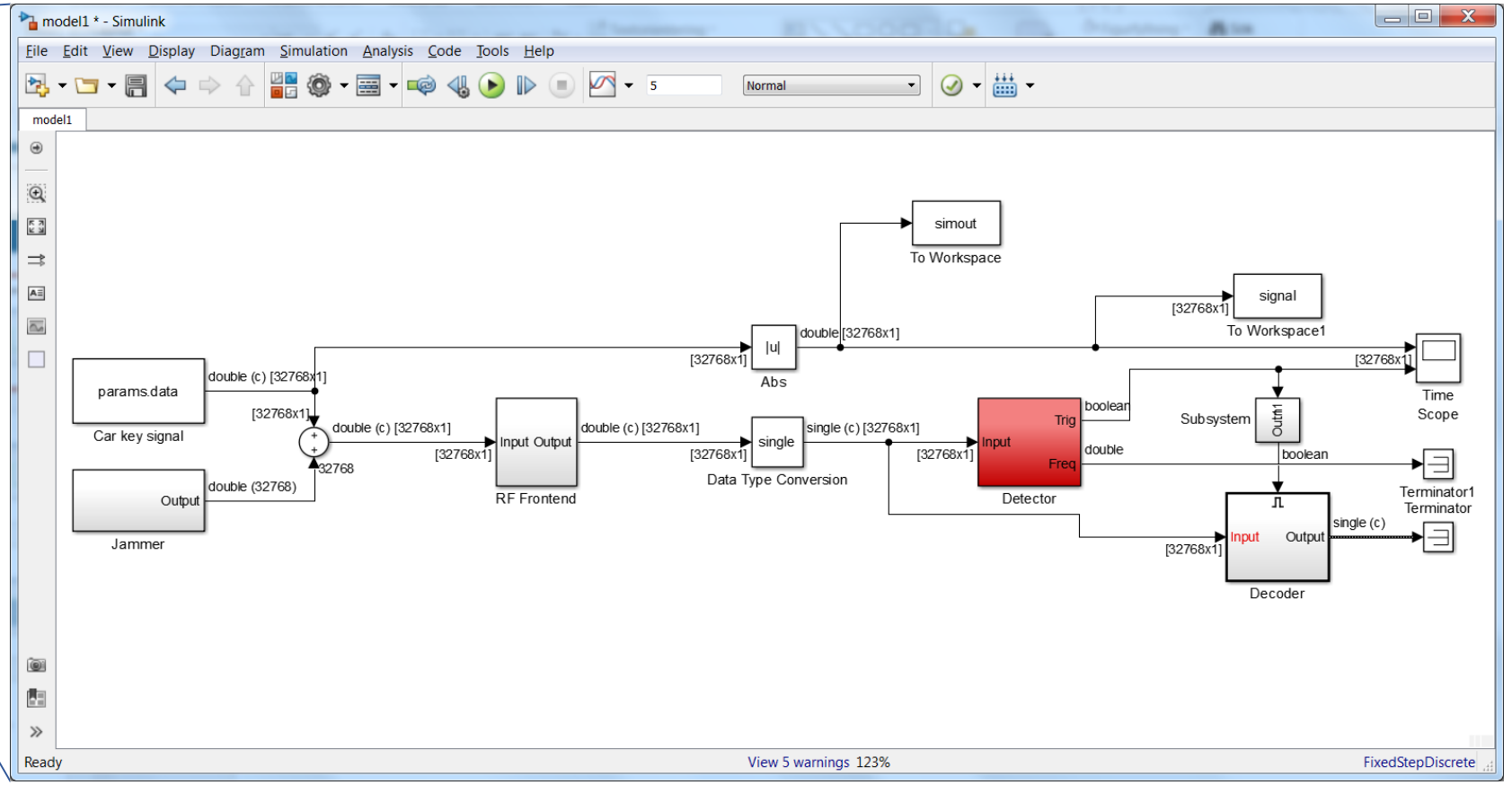
A system model



System level design



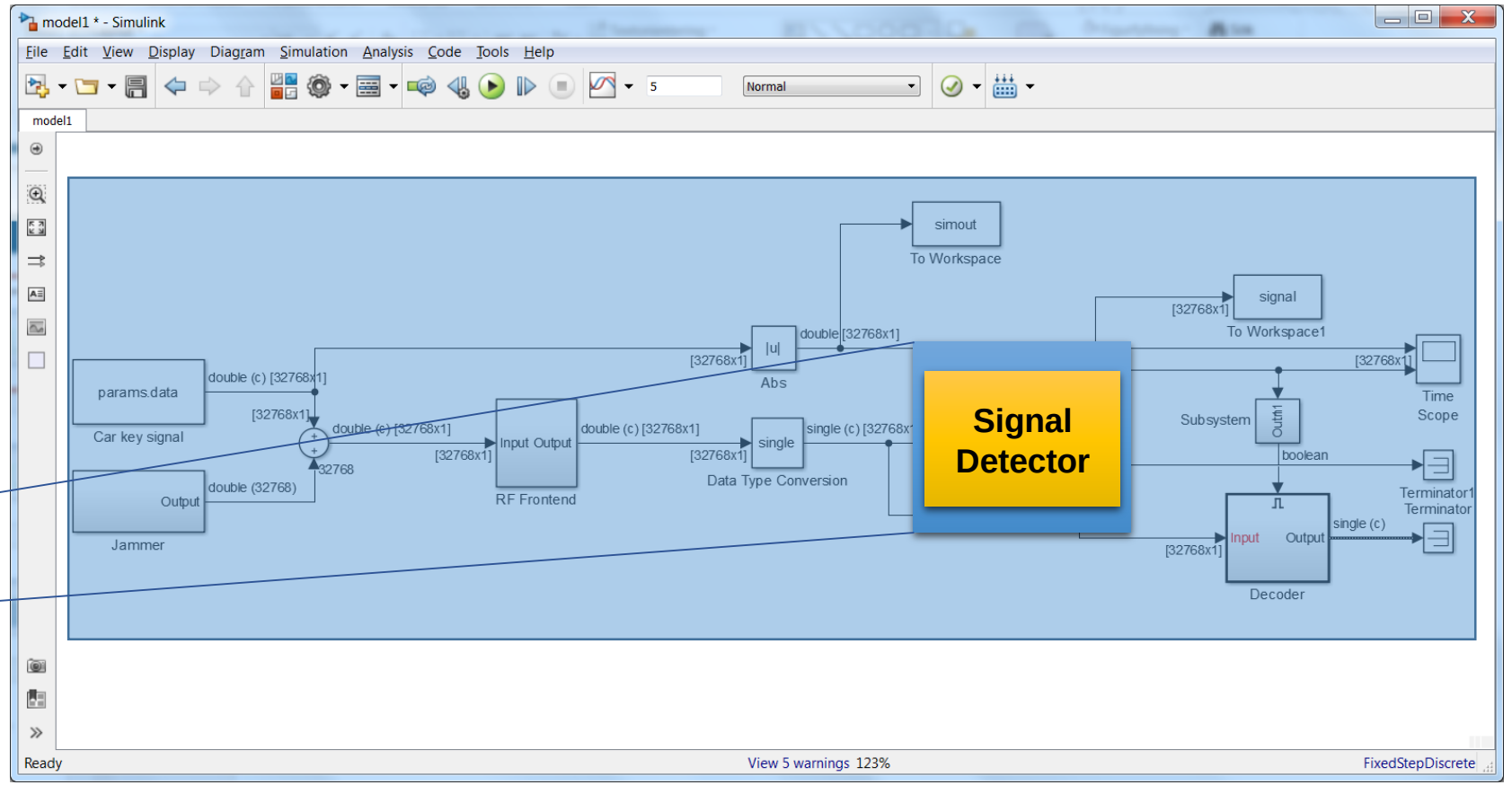
System architect



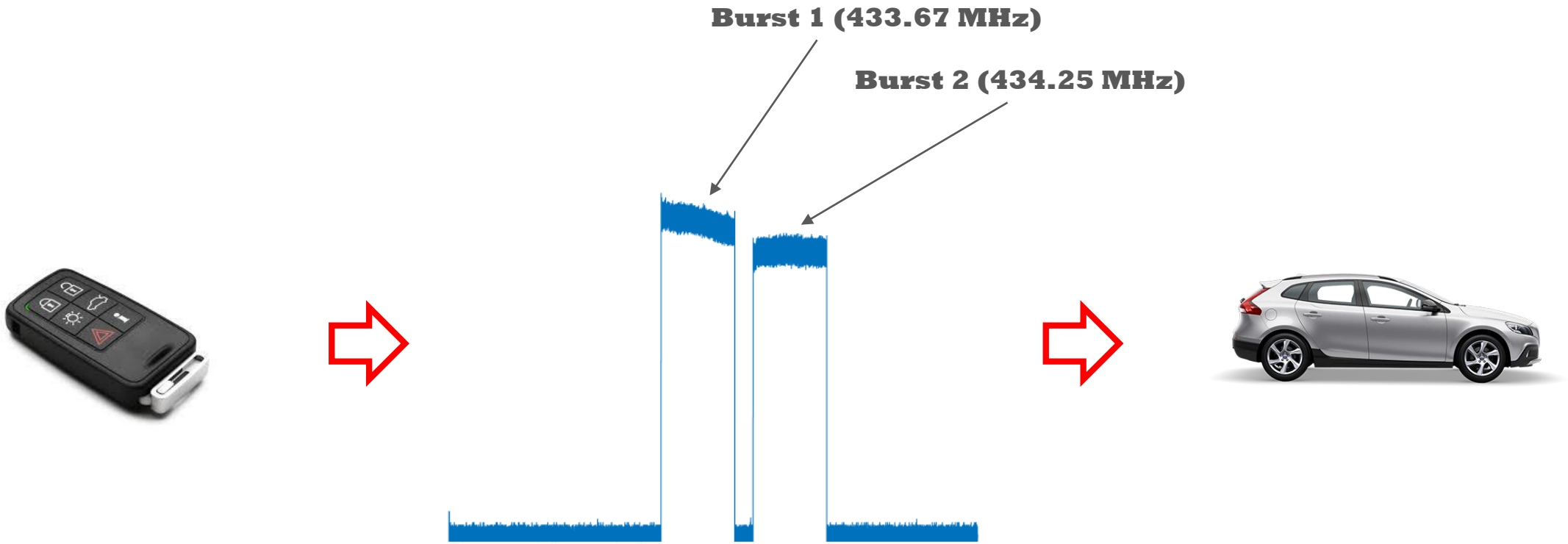
Algorithm design

System architect

Algorithm designer



A toy example



Building an algorithm

- Need to listen for a specific tone (FFT?)
- Probably need some cleaning up (filtering?)
- Need to output a frame of data adequate for the decoder

- Use a buffer for the output. Fill it with the input (filtered).
- Take the FFT of a snippet of the buffer to listen for that tone.

- One FFT, one filter, one buffer
- Constants (buffer lengths a.s.o)
- Initially, reserve buffer memory, set up filter

A traditional way of working

```
clear
close all

load myCapturedData.mat
FrameDuration = 150e-3;
N = 2^floor(log2(FrameDuration*fs));
T = N/fs;  % Frame duration
fc = 433.7e6;

SampleRate = fs;
BufferDuration = 250e-3;
MaxScanDuration = 10e-3;
Threshold = 100;
PassbandFrequency = 50e3;
StopbandFrequency = 75e3;

warning off

FrameLength = N;
BufferLength = ceil(BufferDuration*SampleRate);
ScanLength = 2^floor(log2(MaxScanDuration*SampleRate));
Buffer = complex(zeros(BufferLength, 1));
sLPF = dsp.LowpassFilter('SampleRate', SampleRate, ...
    'PassbandFrequency', PassbandFrequency, ...
    'StopbandFrequency', StopbandFrequency);

numDet = 0;
ii = 1;
```

```
while N*ii <= length(x)
    u = x(N*(ii-1)+1:N*ii);
    nn = BufferLength - FrameLength;
    Buffer(1:nn) = Buffer(end-nn+1:end);
    Buffer(nn+1:end) = step(sLPF, u);
    x1 = complex(Buffer(1:ScanLength));
    X = fft(x1);
    Xpow = real(X).^2 + imag(X).^2;
    Emean = mean(Xpow);
    [Emax, i] = max(Xpow);
    T = ScanLength/SampleRate;
    if i <= ScanLength/2
        f = (i-1)/T;  % Positive frequencies (first half of vector)
    else
        f = (i-ScanLength-1)/T; % Negative frequencies (second half of vector)
    end
    if Emax > Threshold*Emean && (f~=0)
        trigger = true;
    else
        trigger = false;
    end

    spectrum(Buffer)
    drawnow
    if trigger
        numDet = numDet + 1;
        disp(['Detection ' num2str(numDet) ' occured!'])
        disp(['Frequency: ' num2str(f+fc)])
    end
    ii = ii + 1;
end
```

- Data processing is performed on large ”batches” of data
- There’s no separation between algorithm and surrounding test environment (”testbench”)

A traditional way of working

```
clear
close all

load myCapturedData.mat
FrameDuratio  = 150e-3;
N = 2^floor  g2(FrameDuration*fs));
T = N/fs;    ame duration
fc = 433

Sample
Buffe          0e-3;

warning off

FrameLength = N;
BufferLength = ceil(BufferDuration*SampleRate);
ScanLength = 2^floor(log2(MaxScanDuration*SampleRate));
Buffer = complex(zeros(BufferLength, 1));
sLPF = dsp.LowpassFilter('SampleRate', SampleRate, ...
    'PassbandFrequency', PassbandFrequency, ...
    'StopbandFrequency', StopbandFrequency);

num
ii
```

```
while N*ii <= length(x)
    u = x(N*(ii-1)+1:N*ii);
    nn = BufferLength  FrameLength;
    Buffer(1:nn) = Bu  (end-nn+1:end);
    Buffer(nn+1:end)   LPF, u);
    x1 = complex(Buffe  gth));
    X = fft(x1);
    Xpow =
    Emean =
    [Emax,
    T = ScanLength/SampleRate;
    if i <= ScanLength/2
        f = (i-1)/T;  % Positive frequencies (first half of vector)
    else
        f = (i-ScanLength-1)/T; % Negative frequencies (second half of vector)
    end
    if Emax > Threshold*Emean && (f~=0)
        trigger
    else
        trigge
    end

    spectrum(
    drawnow
    if trigger
        numDet = numDet + 1;
        disp(['Detection ' num2str(numDet) ' occured!'])
        disp(['Frequency: ' num2str(f+fc)])
    end
    ii = ii + 1;
end
```

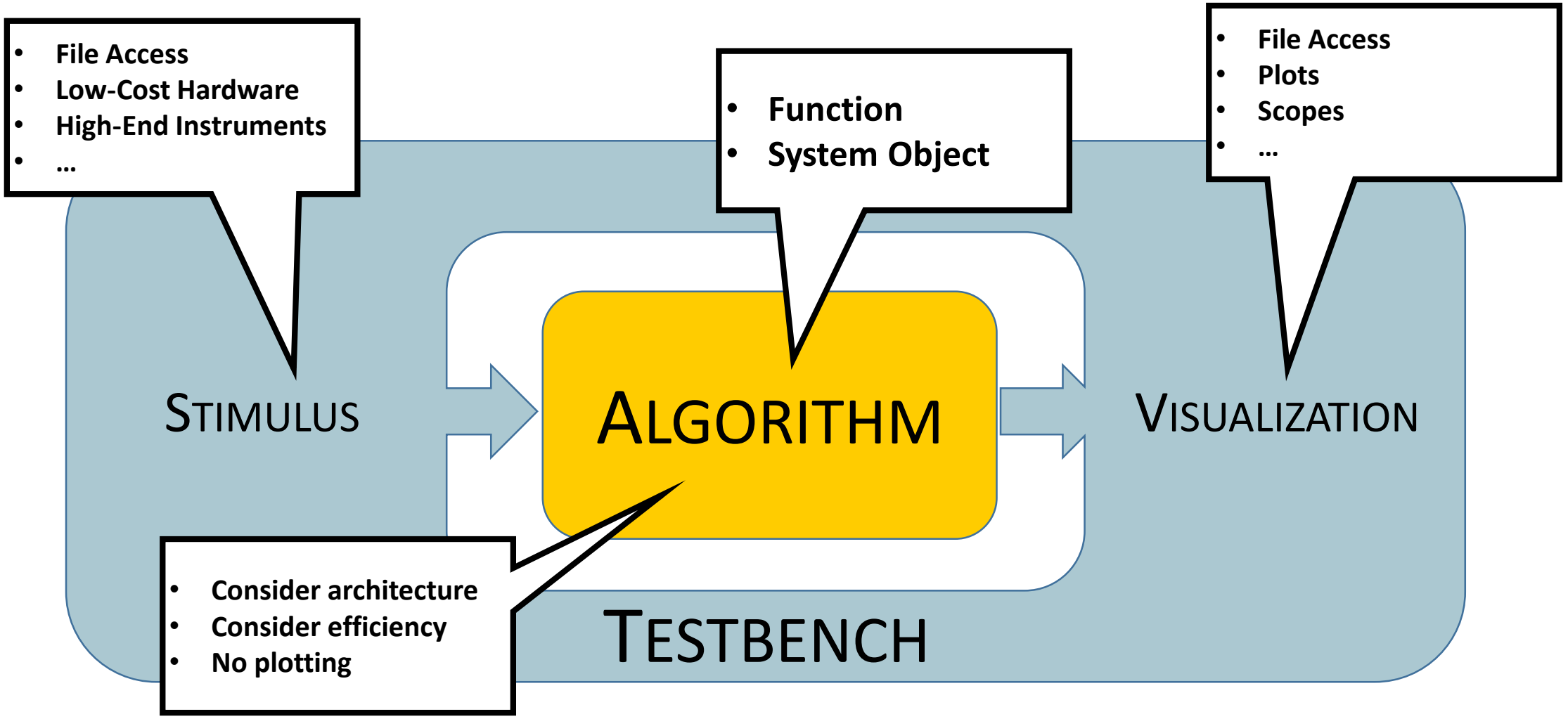
Loading lots of data into memory is inefficient

Manual indexing is error prone

Batch-processing code is hard to convert to a streaming data algorithm!

Continuous plotting using drawnow is slow

Algorithm/testbench separation



System Objects

- Designed specifically for implementing and simulating **dynamic systems** with inputs that change over time
- Use **internal states** to store past behavior, which is used in the next computational step
- Optimized for iterative computations that process large **streams of data**, such as signal processing and audio systems

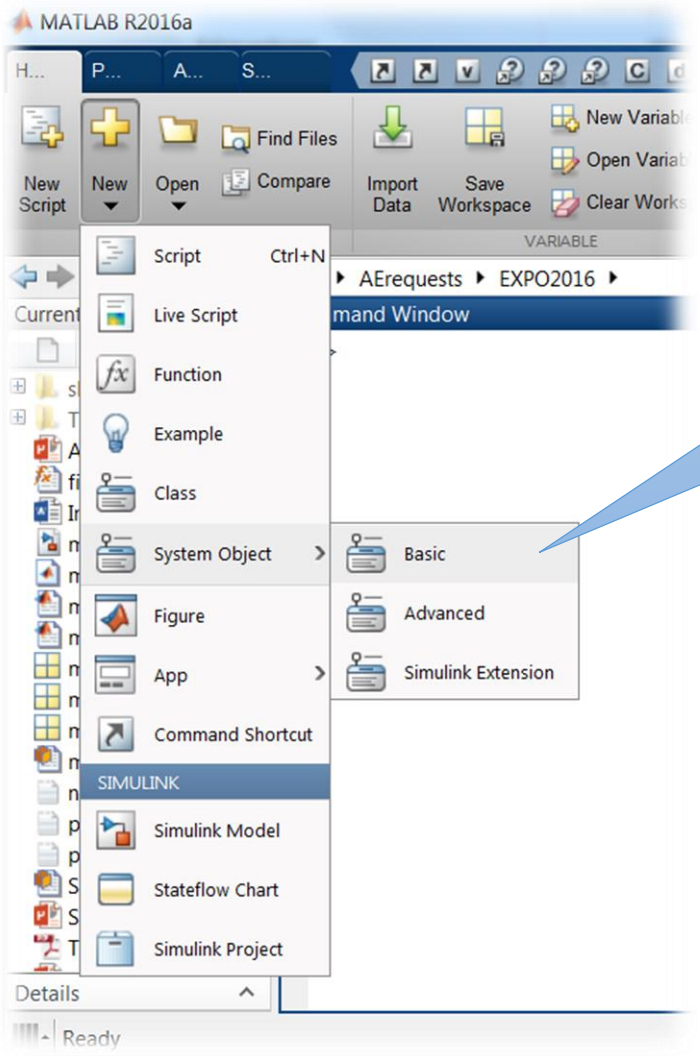
Many System objects support:

✓*Fixed-point arithmetic*

✓*C code generation*

✓*HDL code generation*

✓*Executable files or shared libraries generation*



Access System Object code templates from the menu

- Call *step(<obj_name>, <input>)* to process data
- No need to call *setup()* – it is called automatically the first time you call *step().*
- Public properties are exposed in the Simulink block dialog – access *Preview Block Dialog* from the MATLAB menu

The System Object template

```
classdef Untitled < matlab.System

    properties

    end

    % Pre-computed constants
    properties(Access = private)

    end

    methods(Access = protected)
        function setupImpl(obj)
            % Perform one-time calculations, such as computing constants
        end

        function y = stepImpl(obj,u)
            % Implement algorithm. Calcu  e y as a function of input u and
            % discrete states.
            y = u;
        end

    end
end
```

- One FFT, one filter, one buffer
- Constants (buffer lengths a.s.o)

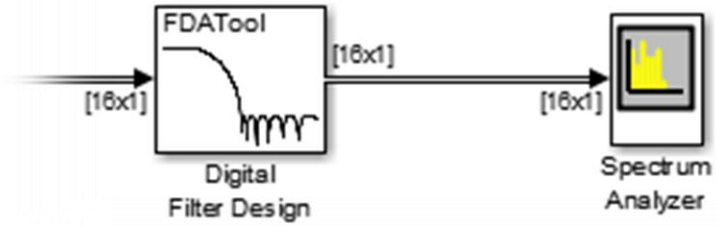
- Initially, reserve buffer memory, set up filter

- Use a buffer for the output. Fill it with the input (filtered).
- Take the FFT of a snippet of the buffer to listen for that tone.

DEMO

Scopes

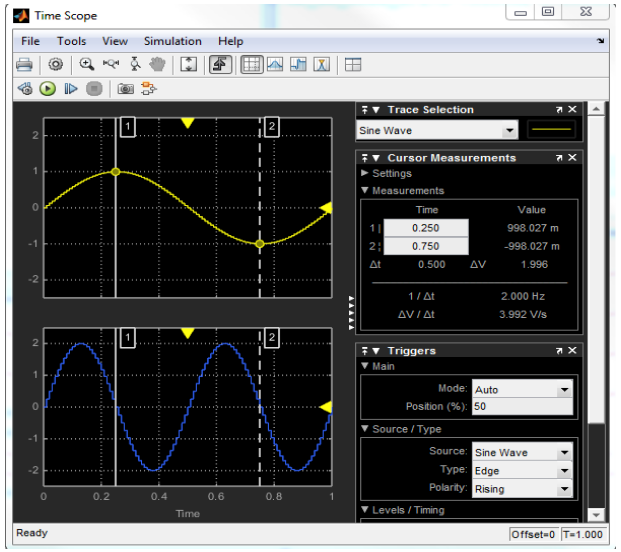
Simulink



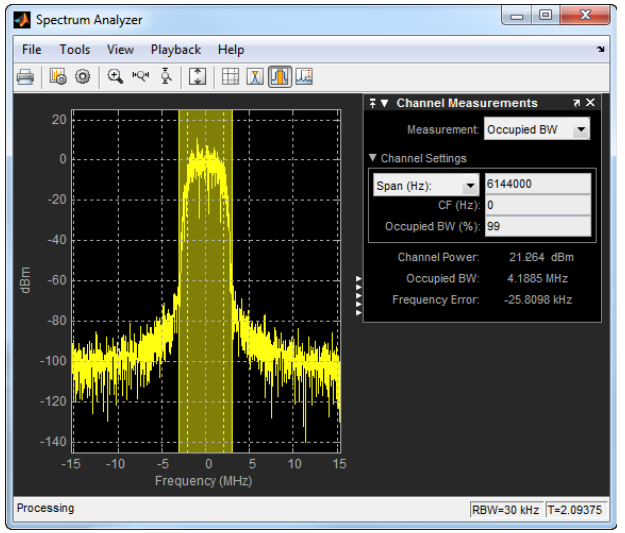
MATLAB

```
» sa = dsp.SpectrumAnalyzer(...
     'SampleRate',10e3);
» …
» step(sa, x);
```

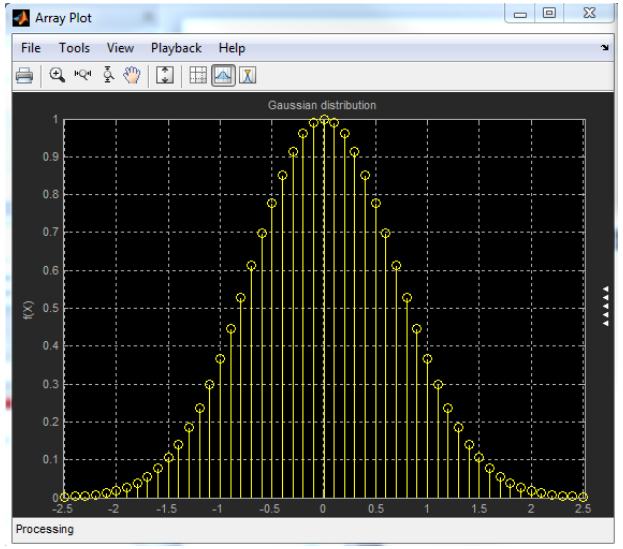
Time Scope



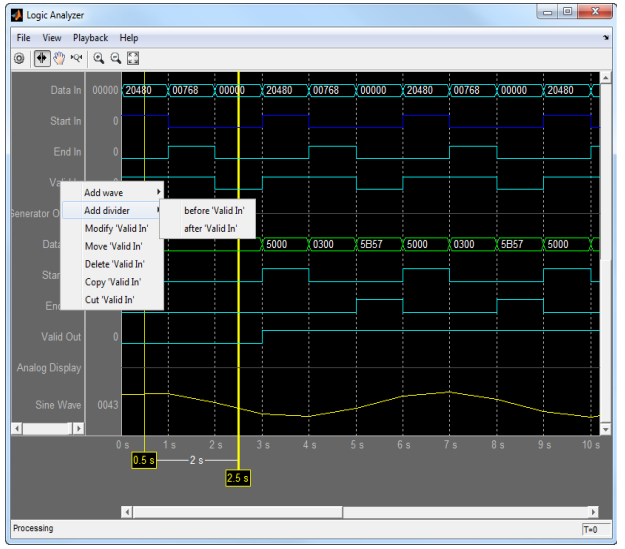
Spectrum Analyzer



Array Plot



Logic Analyzer



Low-Cost Hardware support

DEMO

Summary

- Separate algorithm from testbench!
- System objects provide a framework for efficiently working with streaming data
- System objects provide a seamless way of integrating MATLAB components into Simulink
- Low-Cost Hardware support provides cheap real-world data access