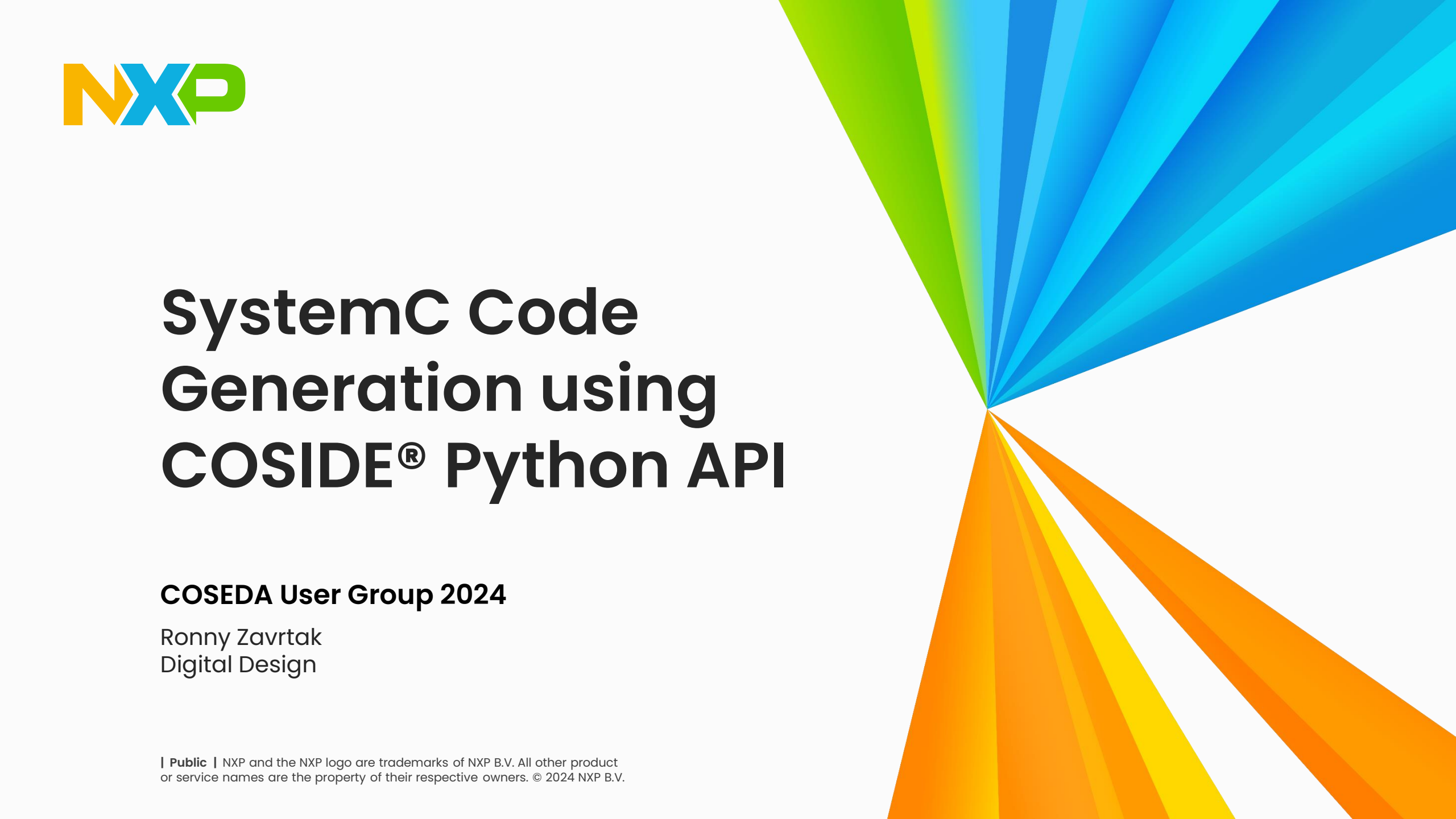
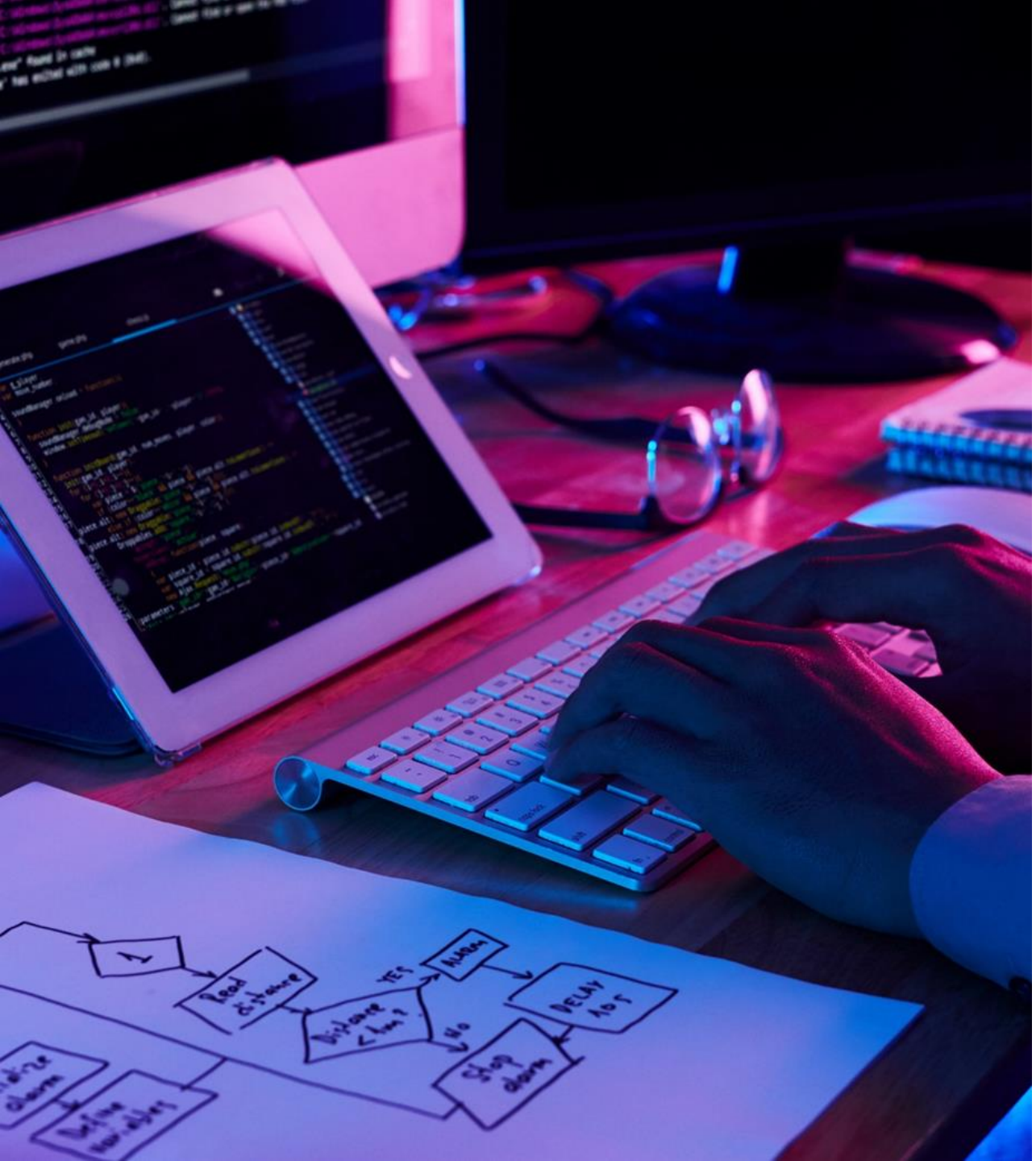


SystemC Code Generation using COSIDE® Python API

COSEDA User Group 2024

Ronny Zavrtak
Digital Design



Navigation section

Agenda

Motivation

Methodology

Model creation

Automation

Code Generation

Flow

Data Model

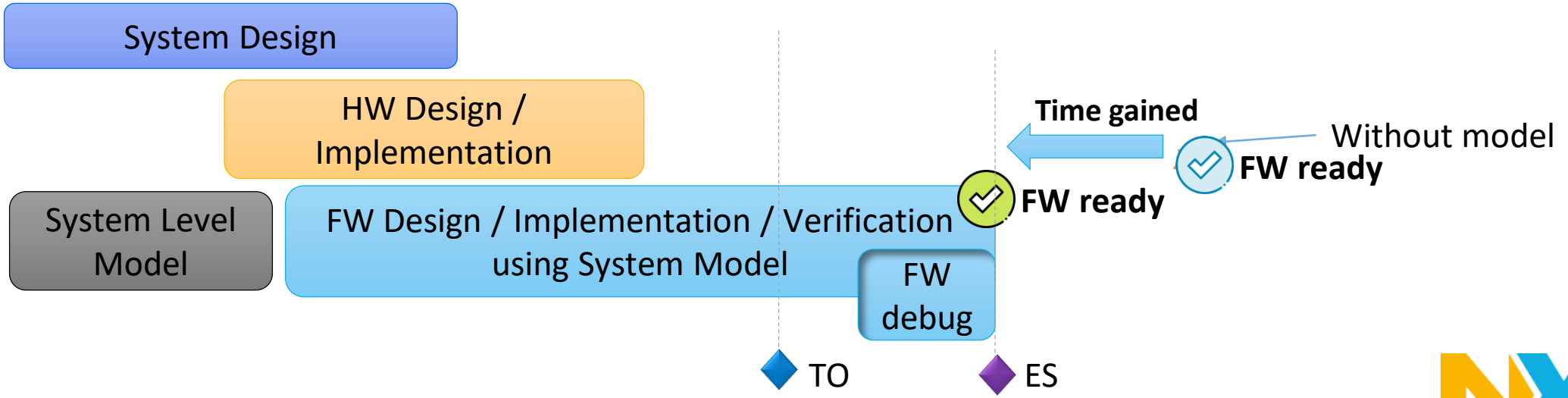
Coside® Features

Python API

Outlook

Motivation

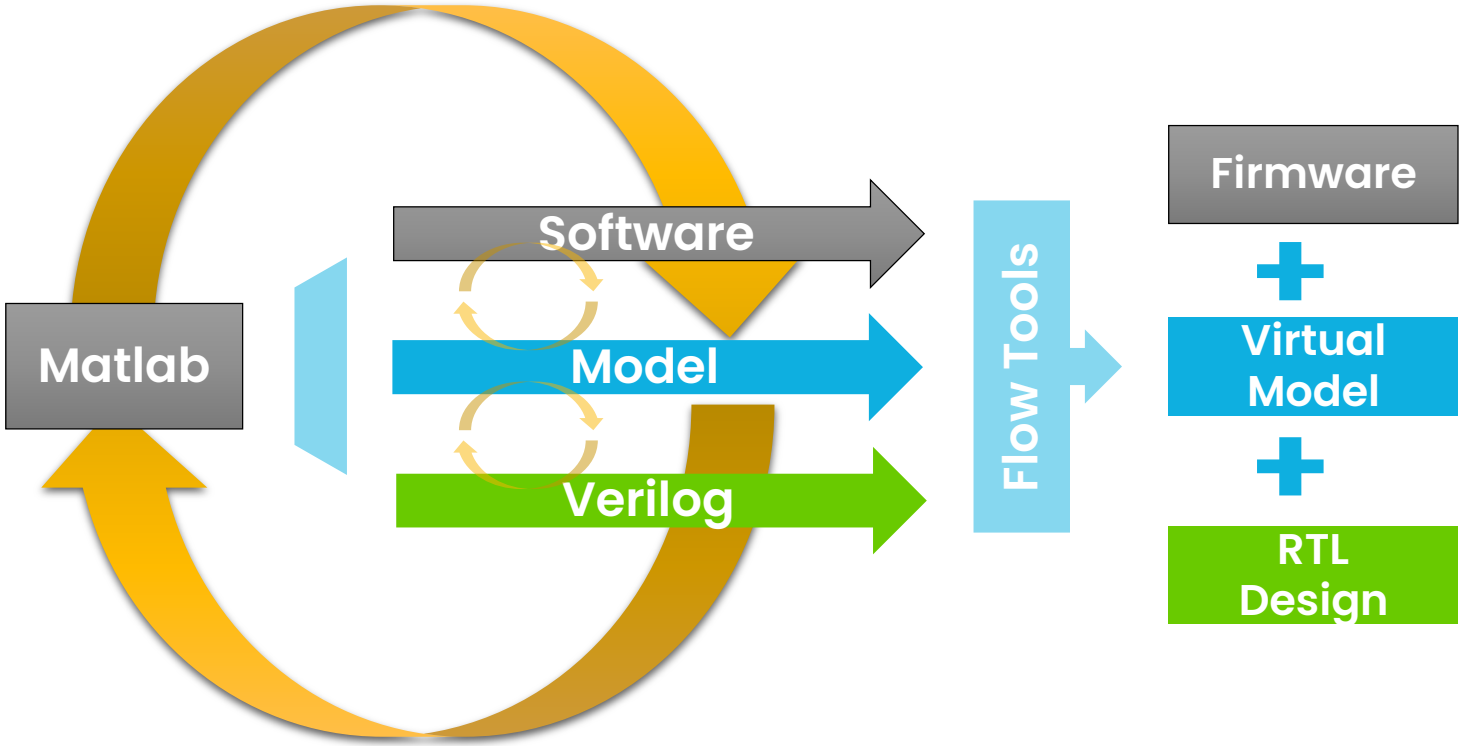
- SW development is complex for large SOCs and requires abstraction to get reasonable speed
  - SystemC simulation up to 10000x faster than RTL simulation
- Timeline and resources are challenging and limited
  - Development Loops
  - Language experts
- SW development needs to start before availability of HW
  - Architecture decisions
  - Complex configurations
- Flexibility in reconfiguration is key for efficient model development
  - Changes only in reference design



Notes and sources

Methodology

- Methodology based on SystemC support hardware and software co-design
- Deliver a higher level of abstraction along with a verification model that accurately reflects the targeted hardware architecture.
- The RTL is always in line with the architectural design
- An agile software development flow unveils very quickly problems in the design



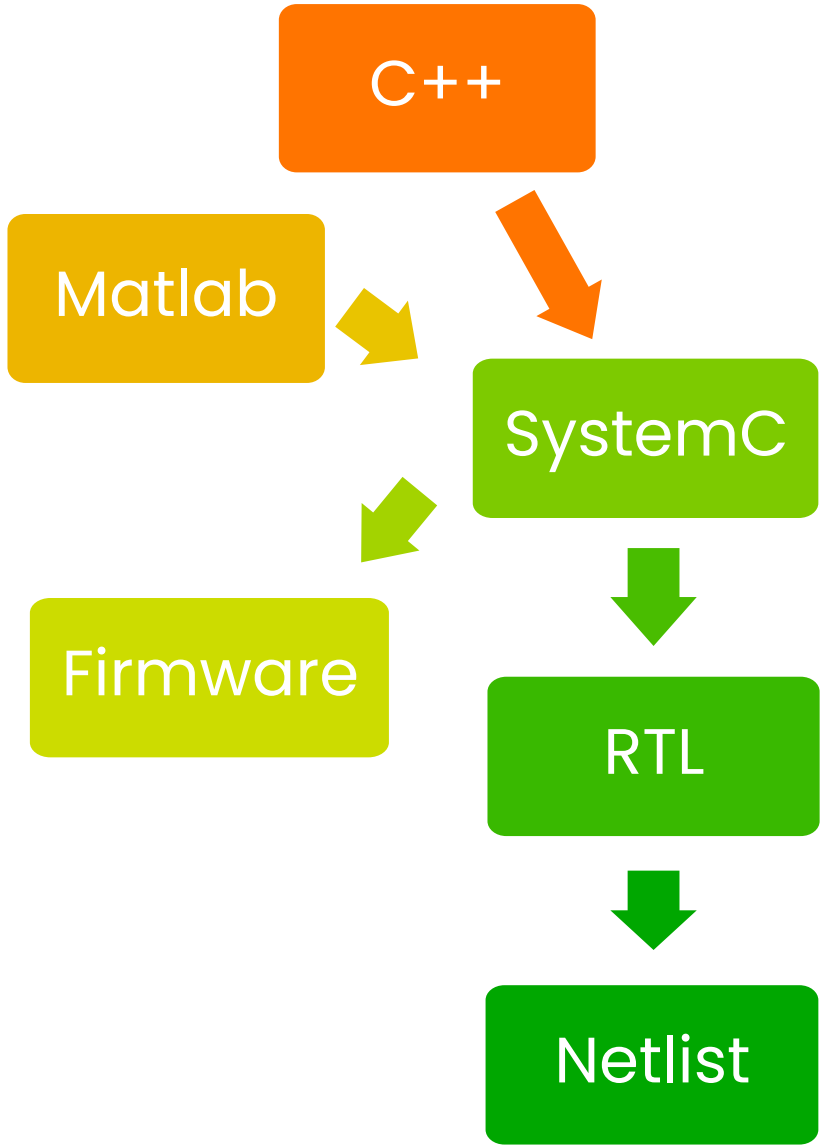
Model to be Created

Analog / RF
- Functional models
- Interface compatible models
- Register compatible models to enable SW development
- Timing accurate model for SW / IC verification

Digital HW
- Convert RTL into SystemC (no Verilator: models not readable, no structure, too late)
- Transaction level models
- Performance

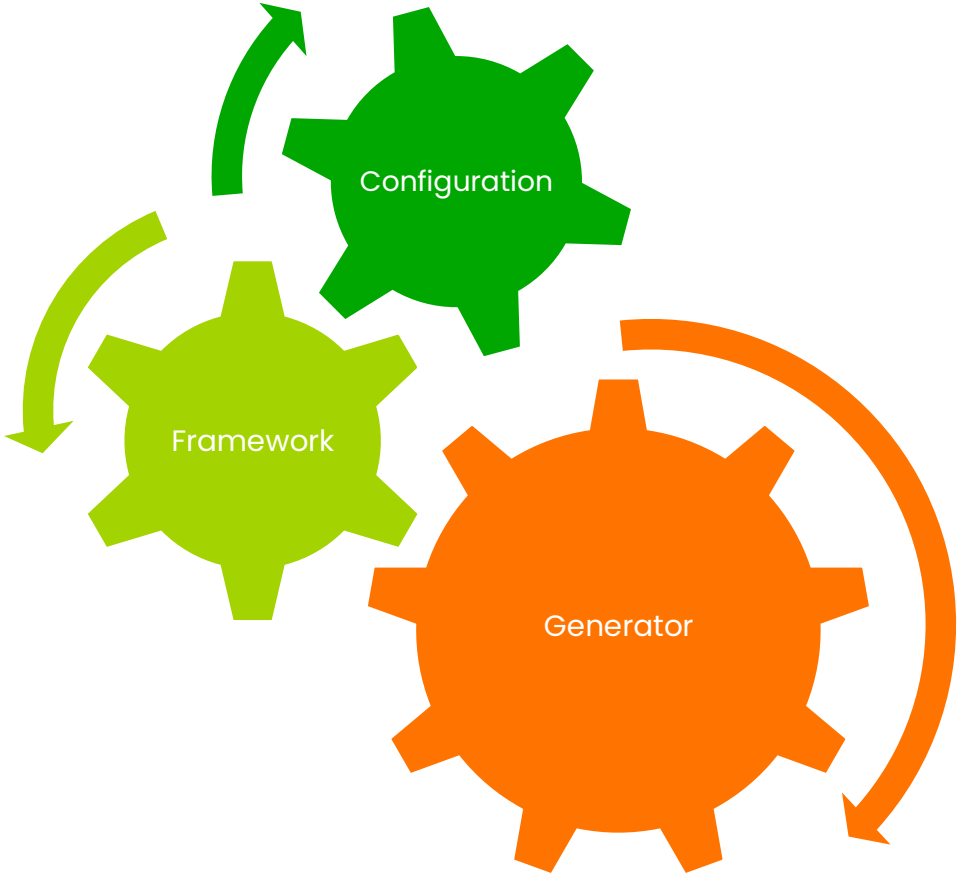
Application:
- Board level models

Integration
- Interconnects and interface definitions
- Register definitions
- Design topologies and IP selection / configurations
- SW API (embedded SW development on RISC Controller)
- Keep design hierarchy throughout project



Automation

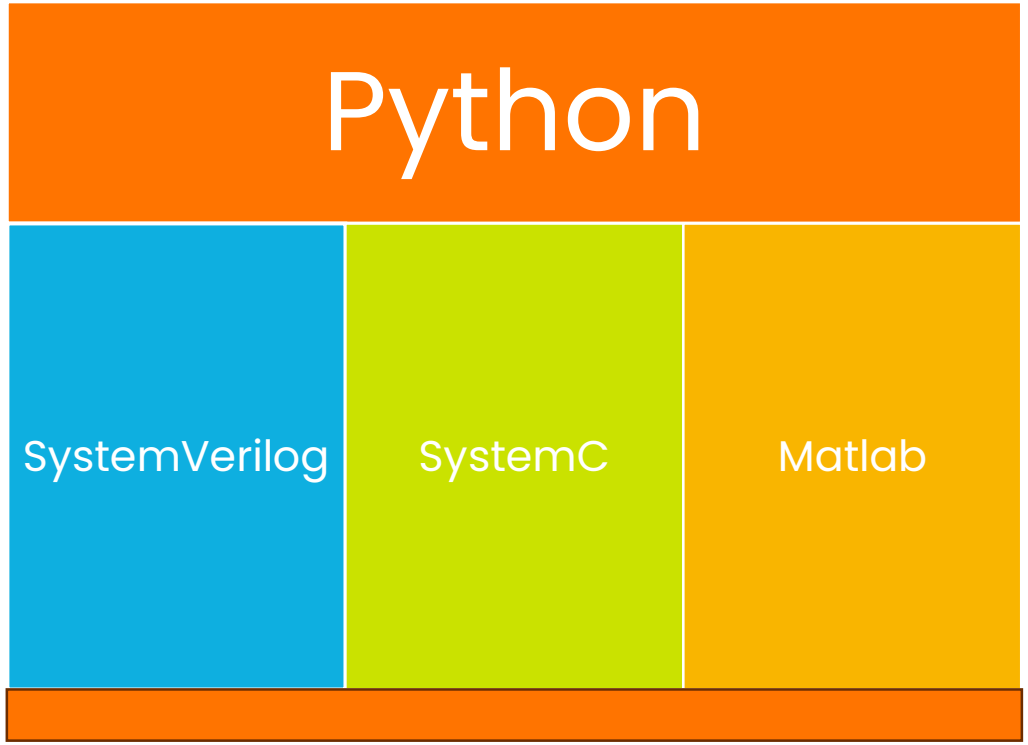
- Flexibility in Configuration
  - Single source
  - Exchange via JSON
- Code generation
  - Structural construction, simulation control and
  - Dynamic model parametrization
- Scripting Environment
  - Use existing libraries
  - Python as well-known and widespread coding language
  - Computational algorithm using numpy/ scipy etc.
  - Advanced features and functionalities like converters and loops
- Common libraries for global components
  - Registers, memories, tracing, infrastructure



Notes and sources

Code Generation

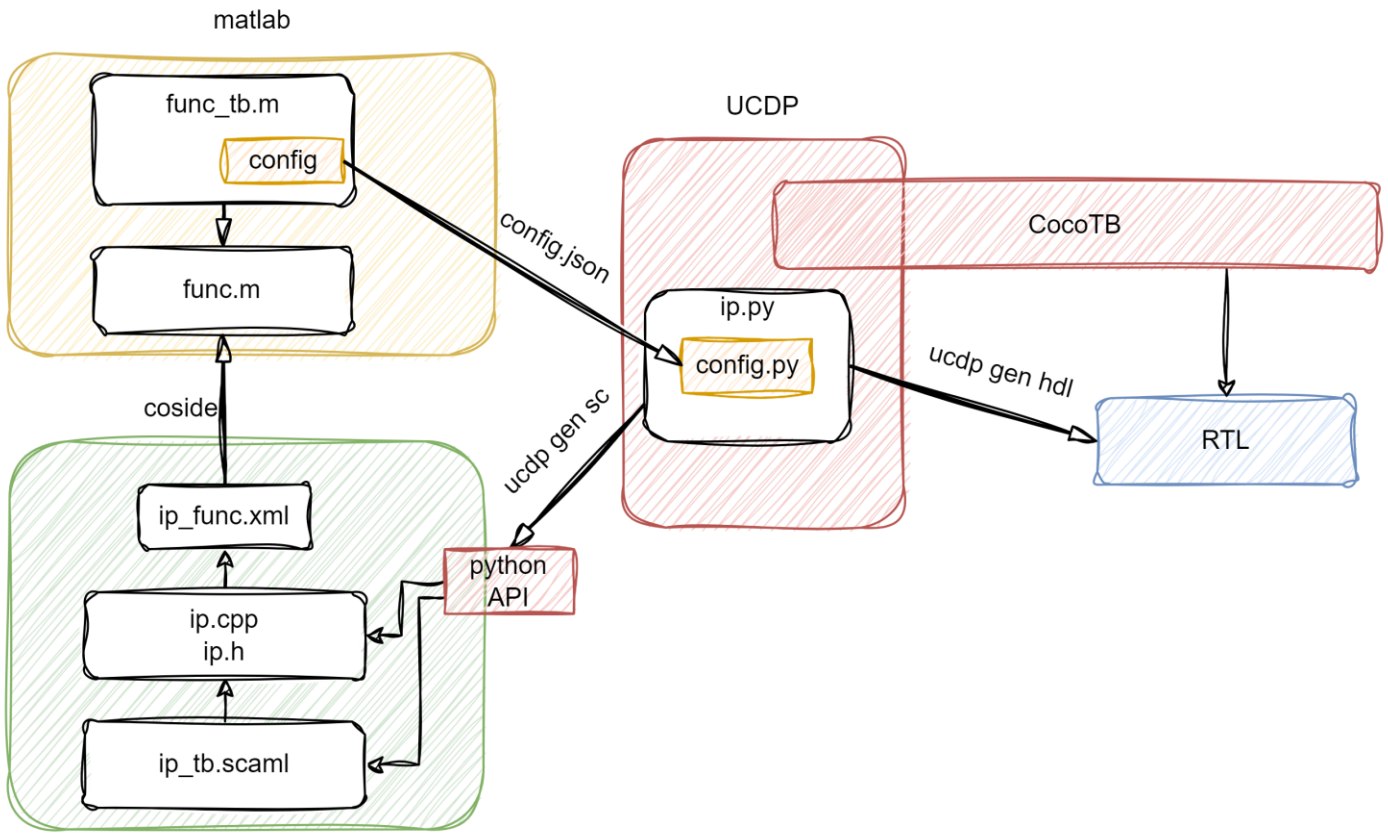
- No duplication of code
  - Interface definition, parameters
  - Behavioral model
- Instantiation of modules and submodules
- Templated and configurable modules
- Named signal and parameter connections
- TLM2.0 support
- Schematic creation
- TB generation
- Trace
- Build
- Run
- Compare



Notes and sources

Flow

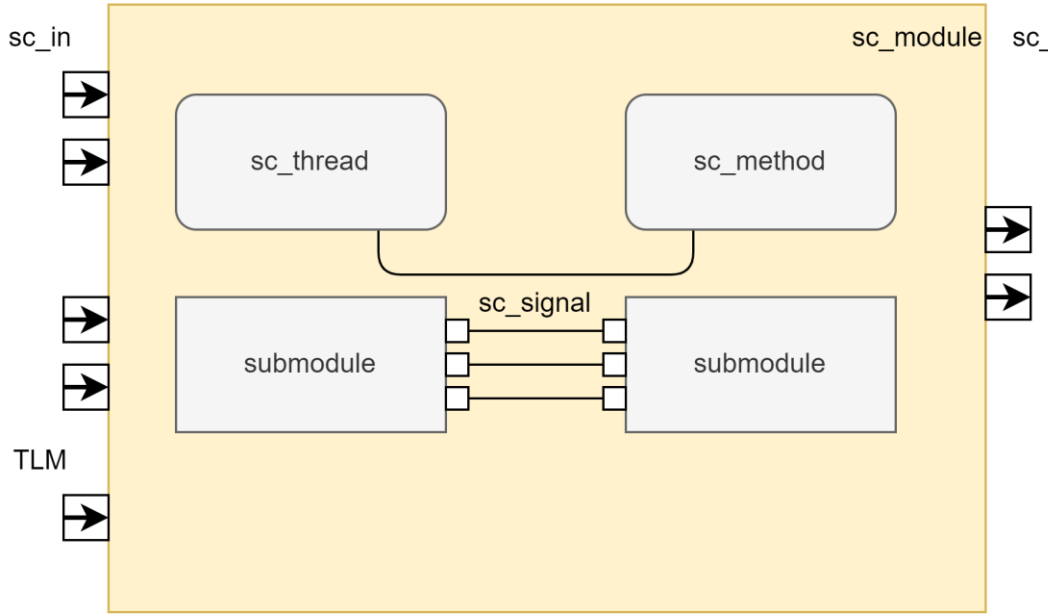
- Matlab reference design configured from testbench
- Config export to python
- Data model
  - Generate RTL stub
  - Generate SystemC model stub
- Matlab reference design verified from Matlab testbench stimuli
  - Verification of RTL design
  - Verification of SystemC model design
- SystemC model call Matlab function



Data Model

- Parametrization
  - Parameter reader
- Ports
  - Type conversion (template parameters)
- Hierarchies
  - Abstraction level
- Bus
  - TLM
- Register file
  - Register mapping
  - Address map
- Signals
  - Interconnections
  - Type conversion
- Configuration
  - Types, bit width, coefficients
- Templates
  - Functional processing

- Functionality is described in a process
- Implemented as member functions
- Code inside a process executes sequentially



Coside® Features*

- Automated code generation
- TLM support
- Offers a wide selection of predefined basic library elements.
- Simple model import and export to numerous tools
- Combination of analog and mixed-signal HW&SW
- Version control

*https://www.coseda-tech.com/systemc-ams-libraries

Coside® Python API

Project	Create Project
	Create Library
	Create Module
Module	Add Parameter
	Add Ports
	Implement Processing
Hierarchy	Instantiate a predefined module
	Named signal connection
Infra-structure	Schematic generation
	Testbench generation
	Build
	Simulation run
	Compare

```
# === import the workspace ========================================================
from cos_scripting import *

# === create a new cos_scripting_example within the specified workspace =======
cos_scripting_example = create_project("cos_scripting_example1")

# === create a new library within the cos_scripting_example ===================
my_lib = cos_scripting_example.create_library("my_lib")

# === create a new module within the library ===================================
my_gain = my_lib.create_sca_tdf_module("my_gain")

# === add ports to the module =================================================
tdf_i = my_gain.add_port("tdf_i", "sca_tdf::sca_in<double>", "dataflow input")
tdf_o = my_gain.add_port("tdf_o", "sca_tdf::sca_out<double>", "dataflow output")

# === add parameter to the module =============================================
my_gain.add_parameter("gain", "double", "1.0", "gain parameter")

# === save module =============================================================
my_gain.save()

# === implement processing() ==================================================
my_gain_loc = my_lib.location() + "/" + "my_gain.cpp"
implement_tdf_processing(my_gain_loc, "my_gain", "tdf_o.write(tdf_i.read()*p.gain);\n")

# === create another new library within the cos_scripting_example ==============
hier_modules = cos_scripting_example.create_library("hier_modules")

# === create a new schematic within the library ================================
sub_hier = hier_modules.create_schematic("sub_hier")

# === define schematic global parameters =======================================
sub_hier.add_parameter("gain", "double", "1.0", "gain parameter")
sub_hier.add_parameter("fc", "double", "1.0", "cut-off frequency")

# === instantiate predefined module ============================================
i_lp_tdf = sub_hier.add_module("$SCA Basic Libraries/analog_filters/lp_tdf", "i_lp_tdf")
i_lp_tdf.set_parameter("fc", "p.fc")

# === instantiate former generated module ======================
i_my_gain = sub_hier.add_module("$Current Project/my_lib/my_gain", "i_my_gain")
i_my_gain.set_parameter("gain", "p.gain")
top.save()

#=== create simple test-bench ================================================
top_simple_tb = top.create_simple_testbench(True)    # default: overwrite=True

#=== compile cos_scripting_example ===========================================
cos_scripting_example.build()

#=== execute test-bench ======================================================
top_simple_tb.run(['--fc=1e3', '--sim_time=5e-3'])
```

Generate.py

```
./coside --cli ../generate.py --workspace ../coside_workspace
```

Fill generate.py from ucdp using **mako** template language

- Query the data model
  - Parameter
  - Configuration
  - Ports
  - Signals
  - Sub-modules
  - Wiring

```
<%!
import ucdp as u
%>

<%inherit file="main.mako"/>

<%block name="main">\
${self.header()}\
${self.footer()}\
</%block>

<%def name="fileheader()">\
<% overview = mod.get_overview() %>\
//
// Module:     ${mod.libname}.${mod.modname}
// Data Model: ${mod.get_modref()}
//
</%def>

<%def name="beginmod(wirenames=None)">\
<%
rslvr = usc.get_resolver(mod)
params = rslvr.get_paramdecls(mod.namespace, indent=2)
ports = rslvr.get_portdecls(mod.ports, wirenames=wirenames, indent=2)
portssignals = rslvr.get_portnames(mod.portssignals, wirenames=wirenames, indent=2)
%>\
% if ports:
 { // ${mod.get_modref()}
${ports.get()}
% endif

// Runtime parameters declaration
% if params:
 #( // ${mod.get_modref()}
${params.get()}
)\
% endif

<%def name="footer()">\
#endif
</%def>
```

Outlook

- Support variants resp. abstraction level
- Interface and BUS abstraction
- Use matlab model in systemc function
- Use matlab stimuli for verification

Abstraction Level		Description	Language	Automation
0	Stub	Ports and register	RTL/SystemC	generated
1	Functional	equation	Matlab/SystemC	hand crafted
2	Architectural	Hierarchy and sublevel	Matlab/SystemC	hand crafted
3	Implementation	Bit true version of the circuit	RTL	hand crafted

01

Questions