Building Code Generators for Reuse – Demonstrated by a SystemC Generator

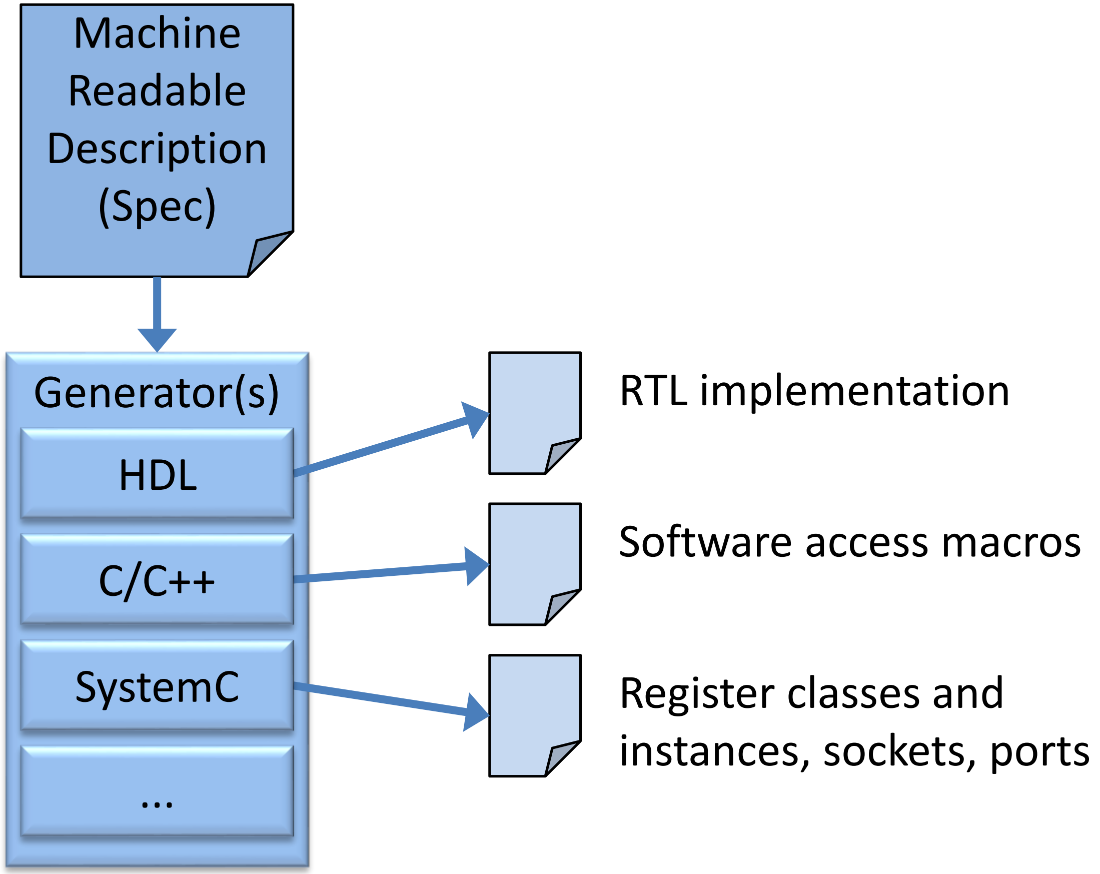
Ulrich Nageldinger, Intel Deutschland GmbH
Saad Siddiqui, Intel Deutschland GmbH

SYSTEMS INITIATIVE
2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

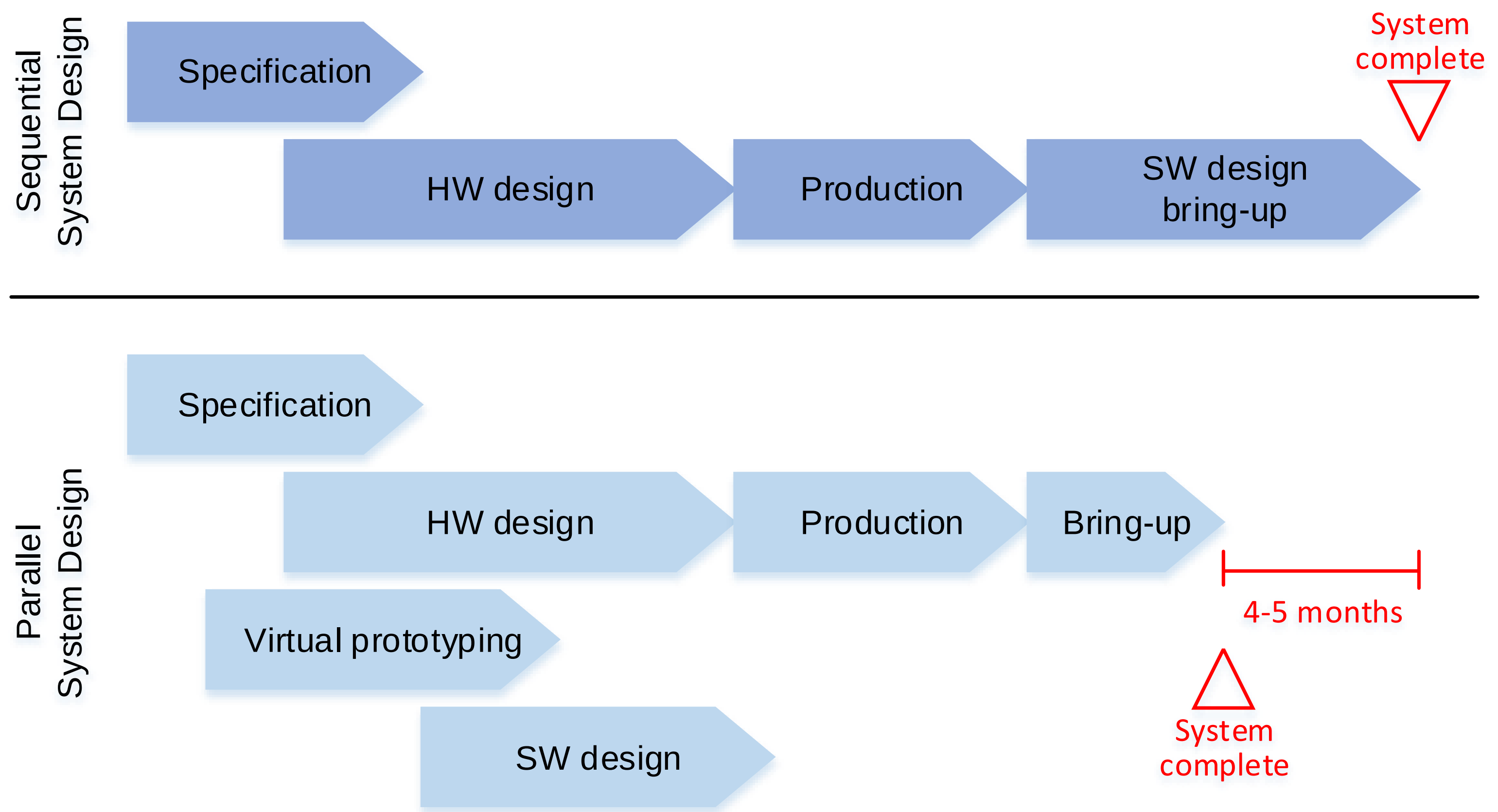
Motivation – Code Generation

- Most popular focus:
  Control register descriptions for bus peripherals
- Automatic code generation easy to implement
  - Machine-readable description
    → (part of) Specification
  - Conversion scripting
  - E.g., Excel-sheet + Visual Basic

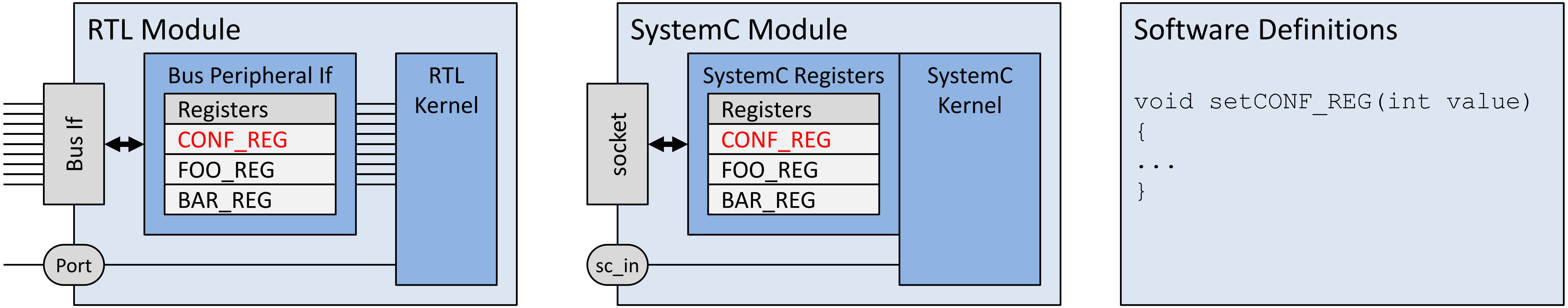
➔ How does it look in a SoC flow?



SoC Design with Virtual Prototype

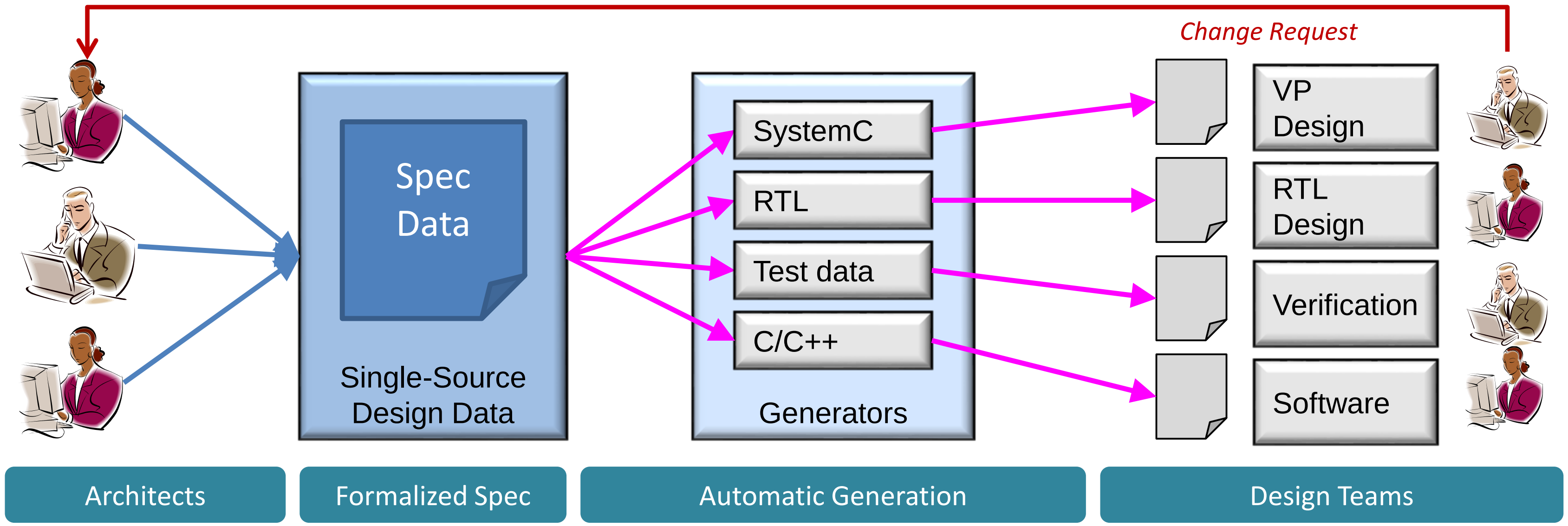


Virtual Prototype Design Requirements



- Requirements for this methodology to work
  - Consistency of collaterals HW ⇔ VP ⇔ SW
  - Fast Propagation of specification changes (→ Bug fixes)
  - Consistent methodology for the whole project, not for one IP

Single-Source Flow for SoC Development



- One single source of formalized specification data
  - Clear ownership
  - Release process
- Build system using generators to propagate spec data to design collaterals

Tooling for Single-Source Flow

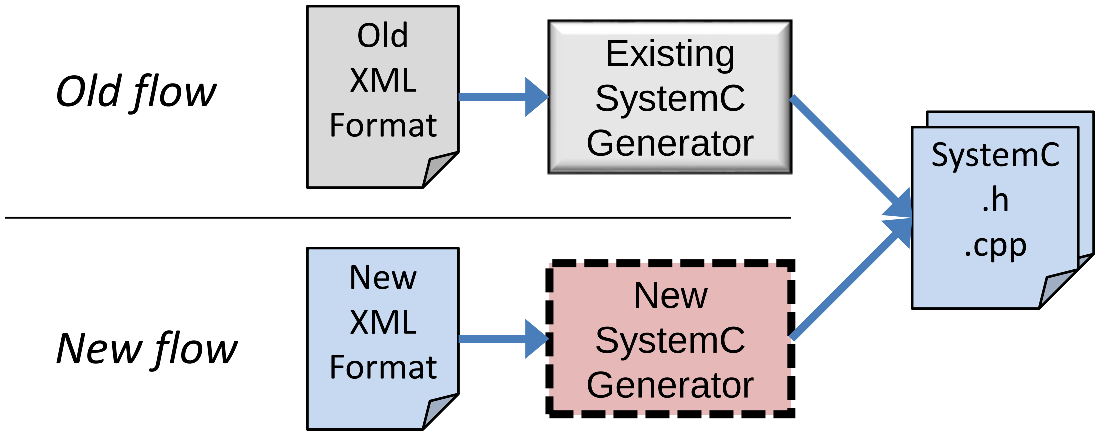
Generators

- SystemC for VP
- HDL for RTL implementation
- C/C++ for Firmware
- C/C++ for Verification
- Register desc. for Debugging
- HTML for internal documentation
- External documentation
- ...

- Single-Source Flow means discipline and effort
  → Want to generate as many collaterals as possible
- *Many* generators needed
  – New tools acquired during project may require new formats
    - Need for fast development of generators
    - Need to enable end-users to write generators
  – Collateral formats may change during project
    - Need for efficient maintainability
  – The underlying data base may change
    - Need for fast migration / reuse of code

Building Code Generators – Case Study

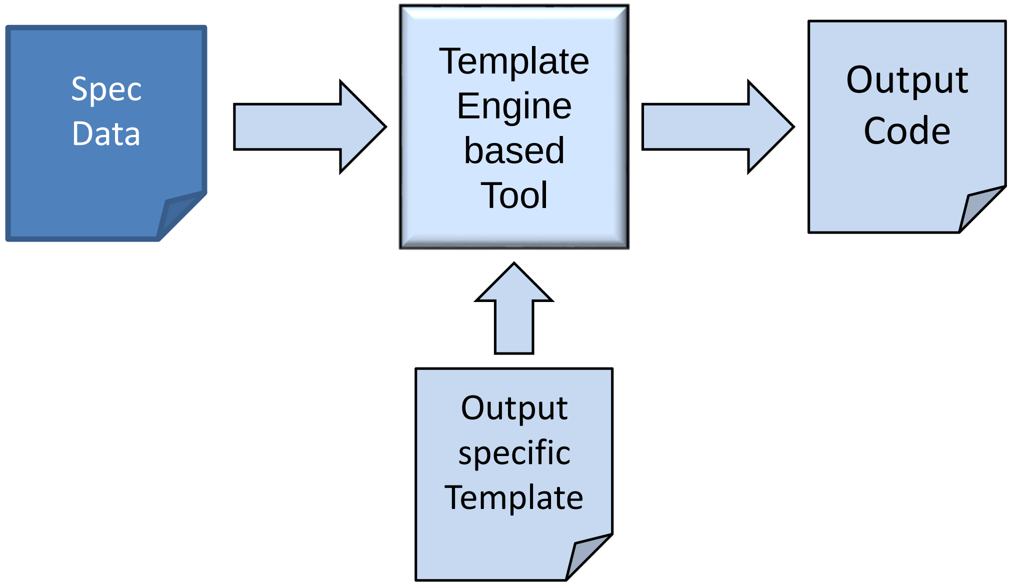
- Situation
  - Existing SystemC generator with a layered architecture
  - Change of underlying data base format (different XML format)
  - Tooling available for both methodologies
- Challenge
  - Get a working solution during running project



General Tooling

- Alternative to:
  - Multiple generator executables
  - One generator executable with hard-coded outputs

➔ Generic Generator tooling with Template Engine
  - Easily extensible (and: end-users can write templates)
  - Good maintainability of templates
  - In case study: Available for both formats



Template Engine

- Well-proven concept for dynamic web page generation
  - Need only one tool (the template engine)
    - E.g., Django, FreeMarker, Mako,...
  - Individual output can be created by different templates
    - Easier to maintain than code
- Works nicely if not too much calculation is needed
  - Model-to-Model transformation needed
  - Still possible to include scripting language code for doing calculation

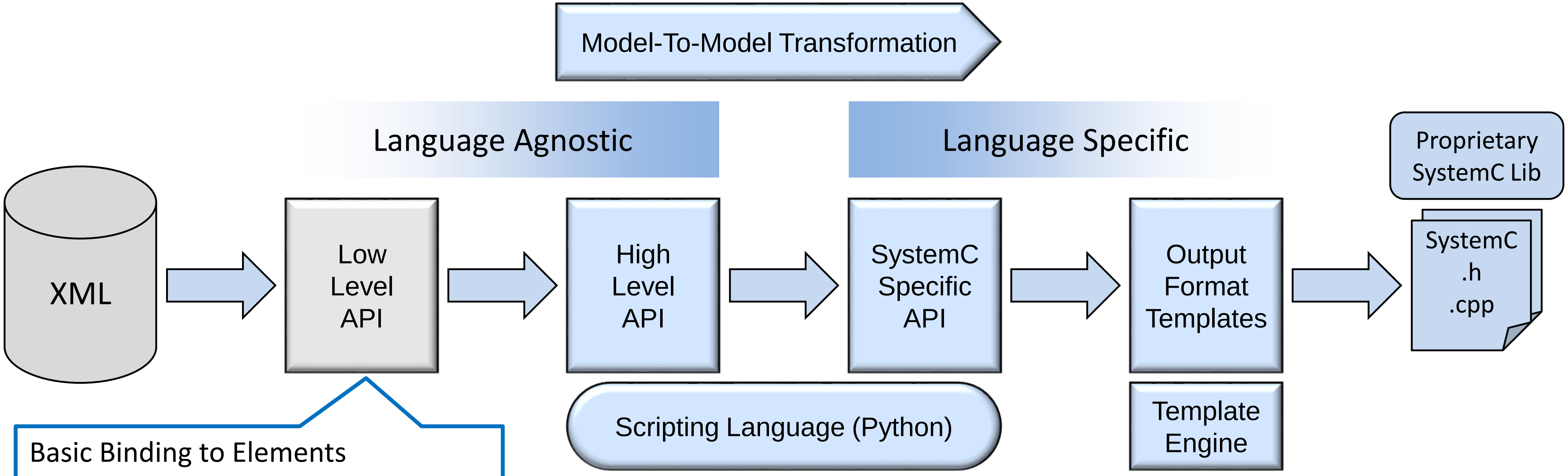
```
print "/* -----------------------"
print "/* Register " + reg.name
print "/* -----------------------"
print "class " + reg.class_name + \
      ":public " + reg.base_name
print "{"
...
```

Using coding

```
/* -----------------------
/* Register ${reg.name}
/* -----------------------
class ${reg.class_name}\
:public ${reg.base_name}
{
...
```

Using template

Generator Architecture - Overview



Basic Binding to Elements

- File handling
- Directly bound to Data Model
- "Get name of register"
- "Get list of fields in register"
- "Get list of interfaces at module"

High-Level API - Overview

- Scripting language (Python) binding to low-level API (in Java or .Net)
  - Easier to use (also by end-users writing their own generators)
  - Dynamic extension of objects (prepare information and attach it to a register)
- Language agnostic, thus reusable for many generators
- Access functions driven by typical use-cases

High-Level API – Example functionality

- Get all physical registers from the input (over hierarchies)
- Get all registers which are visible at a specific bus interface
- Calculate the hierarchical name of a register
- Calculate effective addresses
  (if registers are in nested addrmaps)
- Do filtering of elements
  - E.g., filter out reserved fields

SystemC Context

- Language specific
  - Often not needed for simpler generators
- Example functionality
  - Determine socket type of interfaces (from available types in proprietary library)
  - Determine element class type (from available base classes) and name (by naming convention)
    - E.g., different classes for registers up to 64 bit width and > 64 bit width
  - Determine needed #includes depending on the used elements
  - Use again Python's dynamic object extensibility

Experience for Migration

- Two phases
  - Migration (make it work as before)
  - Improvement (integrate new data model features like better array support)
- Most effort: High-level API
  - Adaptation to underlying data model
    - Different methods to prepare data
  - Refactoring for low-level API functions
  - During improvement phase: high effort to process / prepare the new information
- Medium effort: SystemC context
  - Reused most of the code
  - Improvements for new features required only medium effort
- Low effort: Templates
  - Major reuse
  - New features required some new constructs, but could be easily integrated
    - Convenient maintainability of template-based approach

Results

- Working solution during project with ~4 PW
- Got reusable base for further generator migration
- Due to fast adaptation performance is an area of improvement
  - Mostly due to quick fixes in algorithms

➔ Basic generator methodolgy has proven successful

Thank you for your attention

Questions?