IP-XACT extensions for IP interoperability guarantees and software model generation

Tom Perry[1,2], Richard Walke[1], Rob Payne[1], Stefan Petko[1,2], Khaled Benkrid[2]

[1]Xilinx Scotland [2]ISLI/University of Edinburgh

Introduction

- **Contributions:**
  - IP-XACT extensions
  - Software model generation
- **Outline**
  - Problem statement
  - Proposed solution
  - Metadata details
  - Results and summary

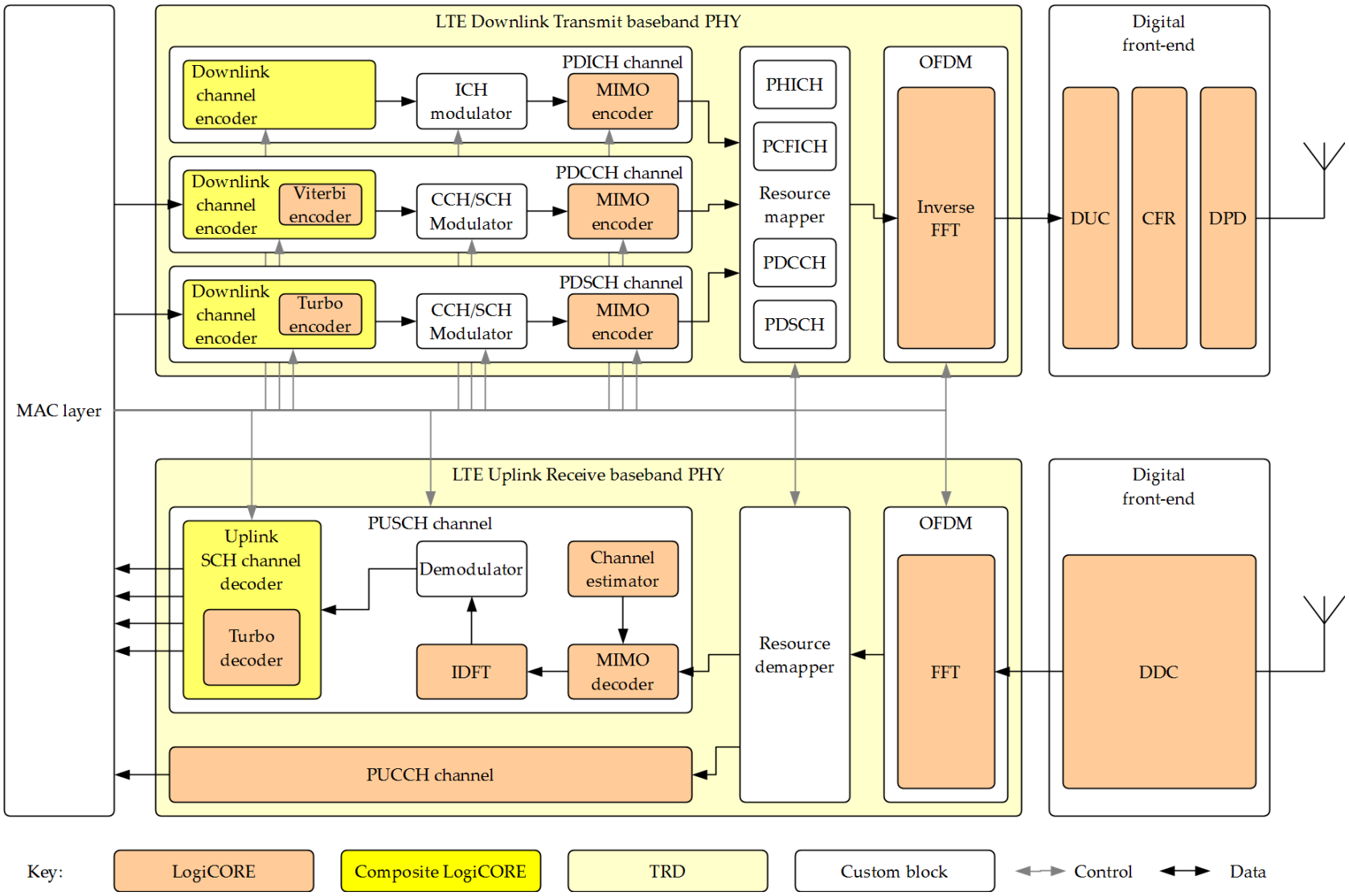
- **Sponsors:**

EPSRC
Engineering and Physical Sciences Research Council

Scottish Enterprise

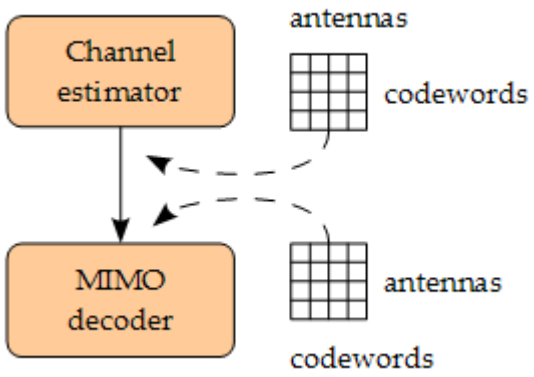
ROYAL COMMISSION EXHIBITION OF 1851

Xilinx LTE physical layer systems



LTE design challenges

- **IP is incompatible at “presentation layer”**
  - Flag errors?
  - Coerce types automatically?

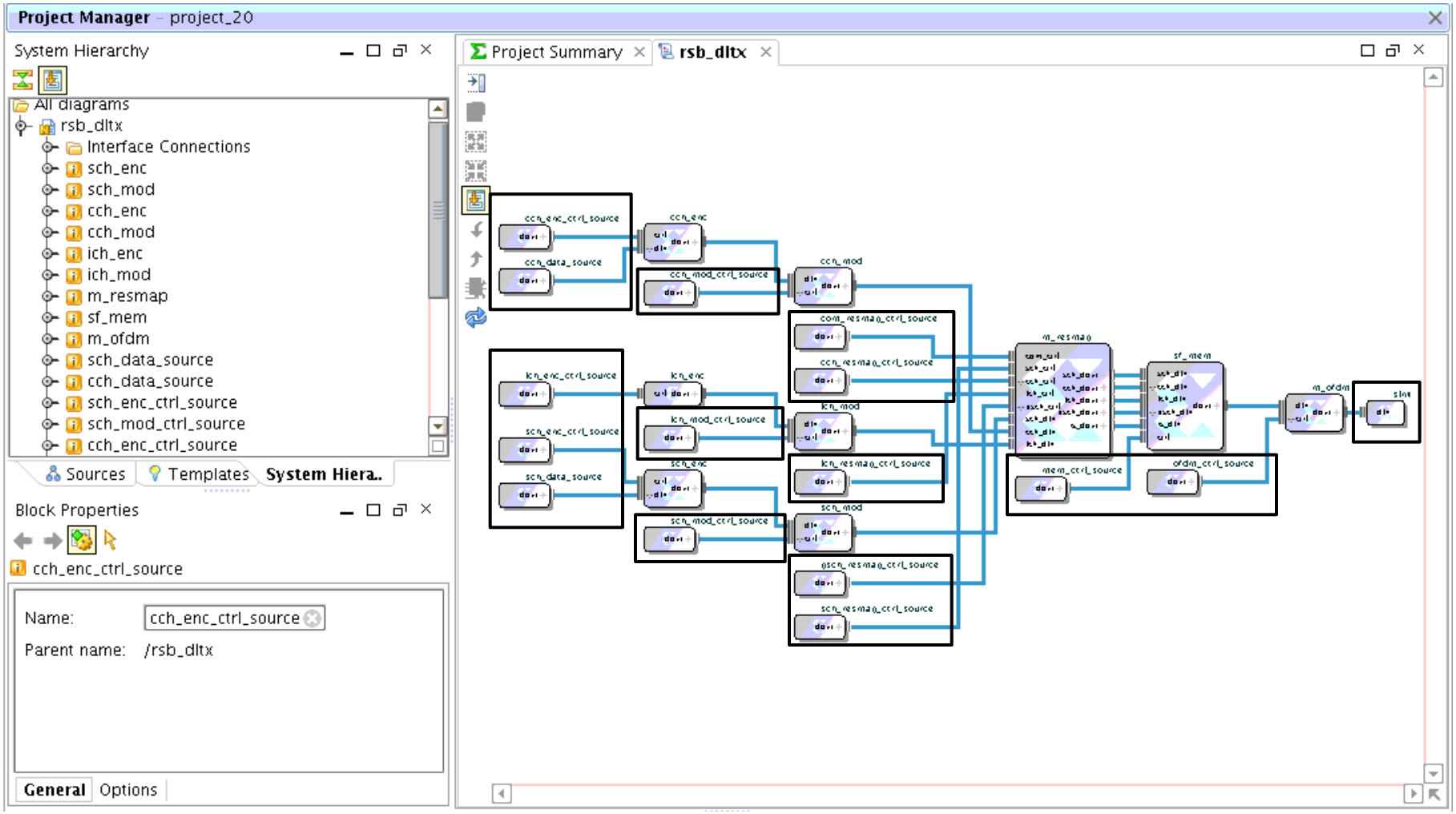


- **System is too complex to implement in HDL from scratch: need a model first**
  - Need to generate test vectors in software which specify required data transactions on hardware block interfaces
  - Model development requires time and effort: 16,000 lines of C++

State-of-the-art in Xilinx

- AXI interfaces on cores – “Plug & Play IP” initiative
- IP-XACT metadata for cores
  - Idea: can we use IP-XACT to describe presentation-layer compatibility?
- Bit-accurate software simulation models for each core
- Vivado IP Integrator
  - Idea: can we use IP Integrator to generate software models?

LTE DL TX system in IP Integrator



Current design flow

HDL

IP Packager

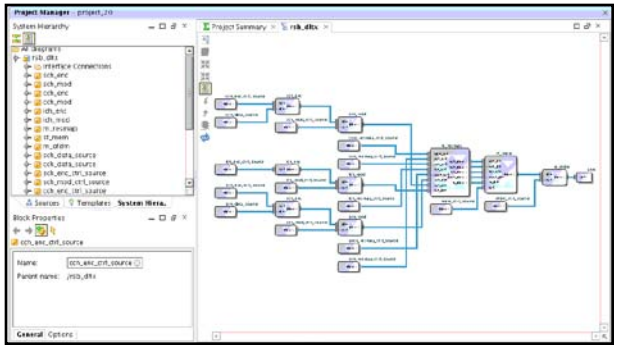
IP Catalog

User HDL

XIP

XIP

IP Integrator design



System with IP, HDL

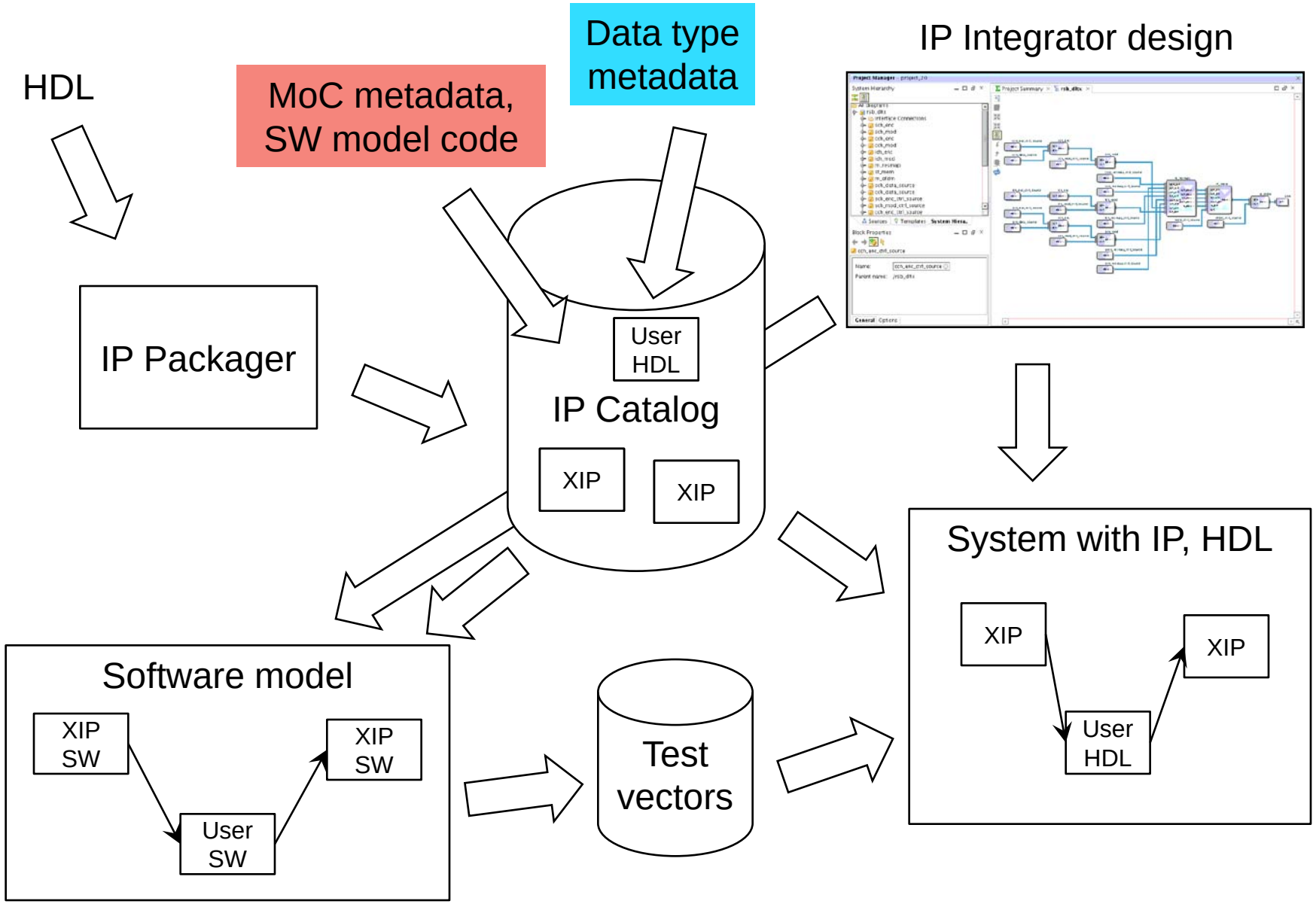
XIP

XIP

User HDL

- **Hard to do system-level architecture exploration**

Our modified design flow



Metadata extensions

- **Aim: extend the IP-XACT metamodel to capture additional aspects of Xilinx IP cores**
- **Data type metadata is needed for two purposes:**
  - Checking IP compatibility
  - Specifying test vector encoding
- **Model of computation metadata is also useful:**
  - Can associate software processing with “actions”
  - Can ensure that latency-sensitive IP cores are not instantiated into a variable-latency system (not discussed in detail here)

Data type storage

```
<spirit:port>
  <spirit:vendorExtensions>
    <x:dataType>
      ...
    </x:dataType>
  </spirit:vendorExtensions>
</spirit:port>
```

- Type description added to a port

```
<spirit:port>
  <spirit:vendorExtensions>
    <x:dataTypeRef
      spirit:vendor="example.com"
      spirit:library="lte"
      spirit:name="resource_block"
      spirit:version="1.0" />
  </spirit:vendorExtensions>
</spirit:port>
```

- Type description stored externally and referenced

Leaf types

```
<x:dataType>
  <x:bitWidth>5</x:bitWidth>
  <x:signed/>
  <x:real>
    <x:fractionalWidth>4</x:fractionalWidth>
  </x:real>
</x:dataType>
```

- **5-bit signed fixed-point type with 4-bit fractional part**
  - Must distinguish width of type from width of container, e.g. 8-bit bus
  - If `<x:bitWidth>` not present, type occupies the full size of the container
- **Similar approach for integers, bools, floats**
- **Complex type is a pair of integers or reals**

Hierarchical types

```
<x:dataType>
  <x:structure>
    <spirit:field>
      <spirit:name>first</spirit:name>
      <spirit:bitWidth>...</spirit:bitWidth>
      <spirit:bitOffset>...</spirit:bitOffset>
      <spirit:vendorExtensions>
        <x:dataType>
          ...
        </x:dataType>
      </spirit:vendorExtensions>
    </spirit:field>
    <spirit:field>
      ...
    </spirit:field>
    ...
  </x:structure>
</x:dataType>
```

- **Structure**
  - Reuses `<spirit:field>` element from IP-XACT register definitions

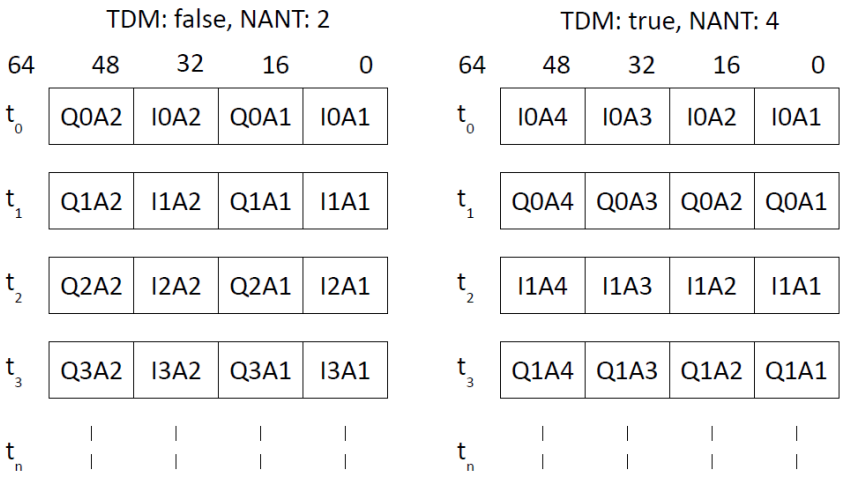
Hierarchical types

```
<x:dataType>
  <x:array>
    <x:name>antennas</x:name>
    <x:size>4</x:size>
    <x:dataType>
      <x:array>
        <x:name>codewords</x:name>
        <x:size>4</x:size>
        <x:dataType>
          ...
        </x:dataType>
      </x:array>
    </x:dataType>
  </x:array>
</x:dataType>
```

- **Hierarchical array**
  - Dimensions are named, allowing incompatible orderings to be detected or corrected
  - Each dimension has a size
  - Dimensions may have a stride…

Parameterisable types

- **Need to handle types that change depending on core parameters**
  - In the DUC/DDC LogiCORE, this can be done with variable stride lengths




```
<x:dataType>
  <spirit:parameters>
    <spirit:parameter>
      <spirit:name>TDM</spirit:name>
      <spirit:value
        spirit:format="integer"
        spirit:id="TDM" />
    </spirit:parameter>
    ... [same for NANT and D_WIDTH]
  </spirit:parameters>
  <x:array>
    <x:name>antennas</x:name>
    <x:size spirit:dependency="id('NANT')"/>
    <x:stride spirit:dependency=
      "if(id('TDM')) then id('D_WIDTH')
                     else id('D_WIDTH') * 2"/>
    <x:dataType>
      <x:complex>
        <x:real>
          <x:bitWidth spirit:dependency=
            "id('D_WIDTH')"/>
        </x:real>
        <x:realInLSBs/>
        <x:stride spirit:dependency=
          "if(id('TDM'))
             then id('D_WIDTH') * id('NANT')
             else id('D_WIDTH')"/>
      </x:complex>
    </x:dataType>
  </x:array>
</x:dataType>
```

Dataflow action metadata

- Each component can be described as an actor with “actions” in CAL language
- C++ processing functions can be scheduled using action metadata
- For Xilinx bit-accurate IP simulation models, we reference [wrapped] software entry point in metadata

```
<x:action>
  <x:input spirit:busRef="din">
    <x:tokenCount>1</x:tokenCount>
  </x:input>
  <x:output spirit:busRef="dout">
    <x:tokenCount>1</x:tokenCount>
    <x:function>
      <x:name>bitacc_simulate</x:name>
    </x:function>
  </x:output>
</x:action>
```

Results

- Data type and MoC metadata can be used to assist in the generation of a software simulation model which outputs encoded test vectors
- Original software models: 16,000 lines of C++ code.
- New models: 3,000 lines of DSL (data types, CAL/NL dataflow) generating 18,000 lines of C++, and 4,000 lines of C++ ‘action functions’ created manually.
- Total code requirement reduced by >50%, C++ requirement reduced by ~75%.

Final comments

- **Data type extensions documented as an IP-XACT “PSS”**
- **Limitations**
  - Highly dynamic types, e.g. IP optional headers
  - Specification of dynamic array sizes
  - Differences between XIP software models → need to add wrappers
- **Ongoing work**
  - Descriptions of data types in IP Packager (Andrew Dow, Xilinx Edin.)
  - Generation of heterogeneous software/hardware systems
  - Mapping components to software (simulation model) or hardware
- **More information available from Stefan and I**
  - Happy to answer any questions