Part B - Agenda

- Review of BridgePoint Modeling Environment
- Introduction to Vista TLM Modeling
  - TLM Modeling and Policies
  - Using Vista TLM ModelBuilder
  - Simulating TLM Models
- Linking BridgePoint UML with Vista TLM
  - The Recipe for exporting and importing models
  - Executing these models in either environment
  - Exploration
- Advanced Topics
  - Incorporating Coverage metrics
  - Modeling Bit-accurate Behaviour

BridgePoint™

Modeling Arithmetic

- Modelers typically start with a PIM where behaviors are captured with infinite precision
- Physical limits are described in the requirements
  - Employed in the model as Range Bound assertions and checks
- Implementation specific characteristics left to Vista domain
  - Likely to explore
    - bit-widths
    - rounding
    - saturation
    - truncation
- Linkage between PIM and PSA is custom datatypes that are marked to appropriate systemC sc_datatypes.

Primer on SystemC Datatypes

SystemC Data Types

- **C++ Built-in Types**
  - **bool - boolean (True or False)**
  - **char - character or integer between -128 and 127**
  - **int - integer**
  - **float * - real number**
  - **strings * - string of alphanumeric enclosed in “”**

* not synthesisable

SystemC Data Types

- Fixed Precision ints
  - 64 Bits
  - `sc_int<N>`
  - `sc_uint<N>`
- Arbitrary Precision
  - Up to 512 Bits
  - `sc_bigint<N>`
  - `sc_biguint<N>`

- Eases modeling of fixed bit-length variables with only truncation
- Most familiar to HW-centric modelers

SystemC Data Types

- **Fixed Point Types**
  - **Useful in DSP applications**
  - **Removes need for error prone translation to equivalent int format.**
  - **Specify Quantization and Saturation Behavior**

  - `sc_fixed<wl, iwl, q_mode, o_mode, n_bits>`

`wl:`	**Total Word Length**
`iwl:`	**Integer Word Length**
`q_mode:`	**Quantization Mode**
`o_mode:`	**Overflow Mode**
`n_bits:`	**Number of Saturated Bits**

SystemC Data Types

- bit and bit vector
  - `sc_bit, sc_bv<N>`
- four-state logic
  - `sc_logic, sc_lv<N>`
    - 0: Logical 0
    - 1: Logical 1
    - Z: High Impedance
    - X: Unknown

- Available for modeling logic and lower abstractions
- Not suitable for PIM or PSA modeling.

SystemC Data Types

- **Common Characteristics**
  - **Native C++ types (`int, float, string`) and SystemC types may be mixed.**
  - **Equality and bitwise operators (==, <<, >>)**
    - **All SystemC Data Types**
  - **Arithmetic and relational operators (+, -, <, >)**
    - **Numeric data types only**
  - **Overloaded assignment operators**
    - **Provides conversion between different data types.**
    - **Conversion may truncate data when necessary.**
      - **e.g.: `myInt = myFloat;`**

SystemC Data Types

- **Utility Methods:**
  - **Bit Select – get and set specific bits**
  - **Range Select – get and set a range of bits**
  - **Concatenation – join bits**
  - **Bitwise Reduction**
  - **Integer Conversion**
  - **String input and output**

SystemC Data Types

- **Bit Select**
  - **Read or write to a specific bit in a variable.**
  - **C++ `operator[]` overloaded to provide read/write access.**
  - **e.g.**

```
sc_int<4> myInt;     // 4-bit signed integer
myInt[1] = true;     // Set bit 1 to true.bool myBool
bool b1 = myInt[0].to_bool(); // Read bit 0
```

SystemC Data Types

- **Part Select**
  - **Read/write a contiguous subset of bits within the variable.**
  - **Available methods:**
    - `range(int, int)`
    - `C++ operator()`
  - **e.g.**

```
sc_int<8> myInt = 2;                      // 00000010
myInt.range(3, 2) = myInt.range(1, 0);    // 00001010
```

SystemC Data Types

- **Concatenation**
  - **Concatenate the bits of two variables together.**
  - **Available methods:**
    - `concat(arg0, arg1)`
    - **C++ comma operator** “`operator,`”

```
sc_int<8> U1 = 2; // 00000010
sc_int<2> U2 = 1; // 01
sc_int<8> U3 = (true, U1.range(3,0), U2, U2[0]);
                  // U3 = 10010011

(U2[0], U1[0], U1.range(7,6)) = U1.range(3, 0);
```

SystemC Data Types

- **Bitwise Reduction**
  - Performs bitwise operation on all bits in integer or vector.
  - Returns bool.
  - Operations:
    - `and_reduce()` - Bitwise AND between all bits
    - `nand_reduce()` - Bitwise NAND between all bits
    - `or_reduce()` - Bitwise OR between all bits
    - `nor_reduce()` - Bitwise NOR between all bits
    - `xor_reduce()` - Bitwise XOR between all bits
    - `xnor_reduce()` - Bitwise XNOR between all bits

SystemC Data Types

- **Integer Conversion**
  - **All SystemC data types**
    - **accept C++ integer assignment.**
    - **convert to C++ interger types**
  - **Conversion Methods:**
    - `to_int()` **- Convert to native int type**
    - `to_uint()` **- Convert to native unsigned type**
    - `to_long()` **- Convert to native long type**
    - `to_ulong()` **- Convert to native unsigned long type**
    - `to_uint64()` **- Convert to native 64-bit unsigned integer**
    - `to_int64()` **- Convert to native 64-bit signed integer**

SystemC Data Types

- **String input and output**

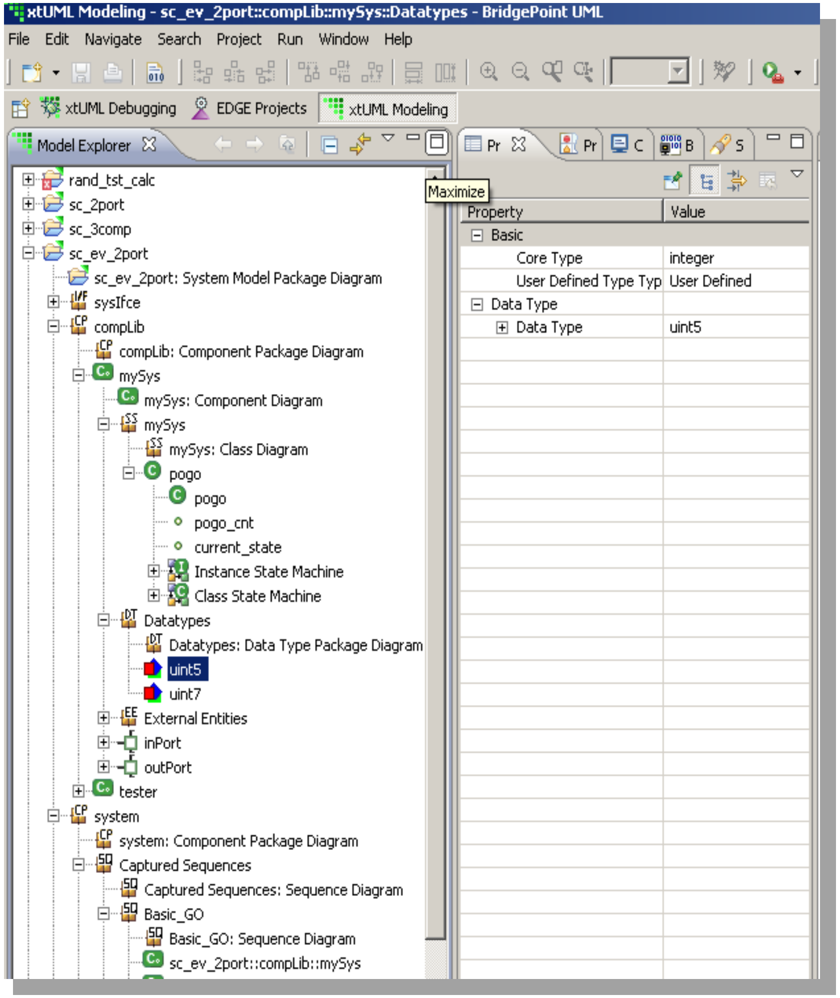
  - **All SystemC data types**
    - **can be set by reading from a C++ input text stream**
    - **can print their value to a C++ output text stream**

```
void scan(istream& input);
void print(ostream& output);
```

Recipe for Bit-Accurate Modeling

Step 1 – Custom Data Types

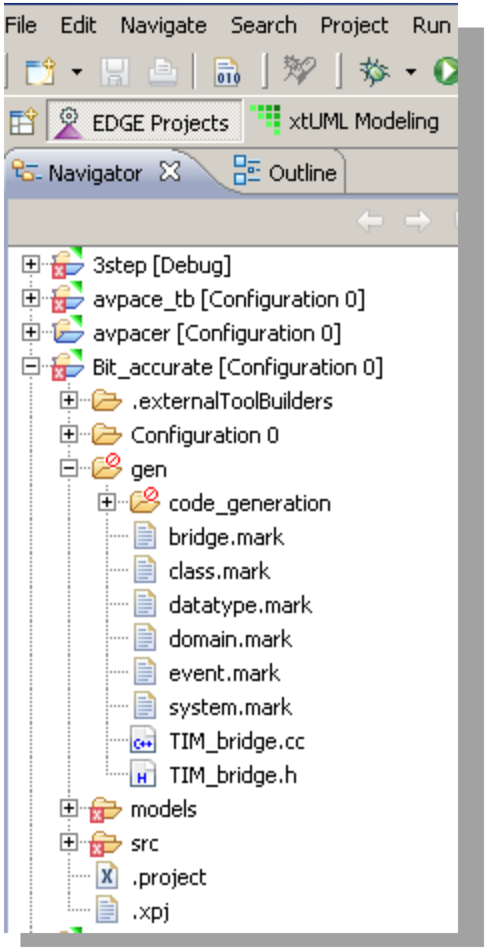
- machine native types
  - Typically choice of target language
  - Typically multiples of 8
- Create user data types
  - Recommended
  - Name must be unique within domain
  - Code generation will translate to systemC datatype



Step 2 - SystemC

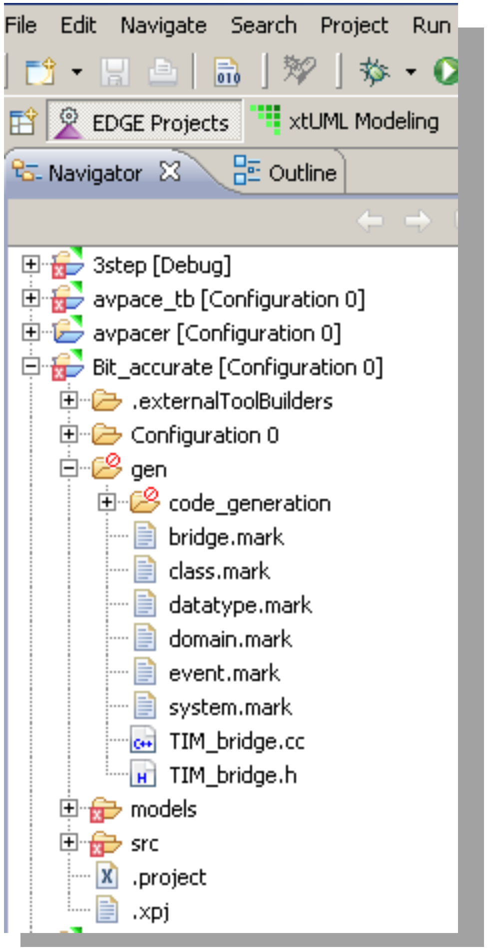
- **SystemC datatypes offer a standardized library of types for modeling hardware**

- **Extends native datatypes to model**
  - **bit-width**
  - **saturation**
  - **rounding**



What about AC Datatypes

- **The AC (Algorithmic C) datatypes are better for synthesis and hardware modeling**
- **AC datatypes are compatible with systemC**
- **BridgePoint will make the substitution in generated code**
- **Requires manual inclusion of ac_int.h and ac_fixed.h**



Step 3 – Defining the Mark

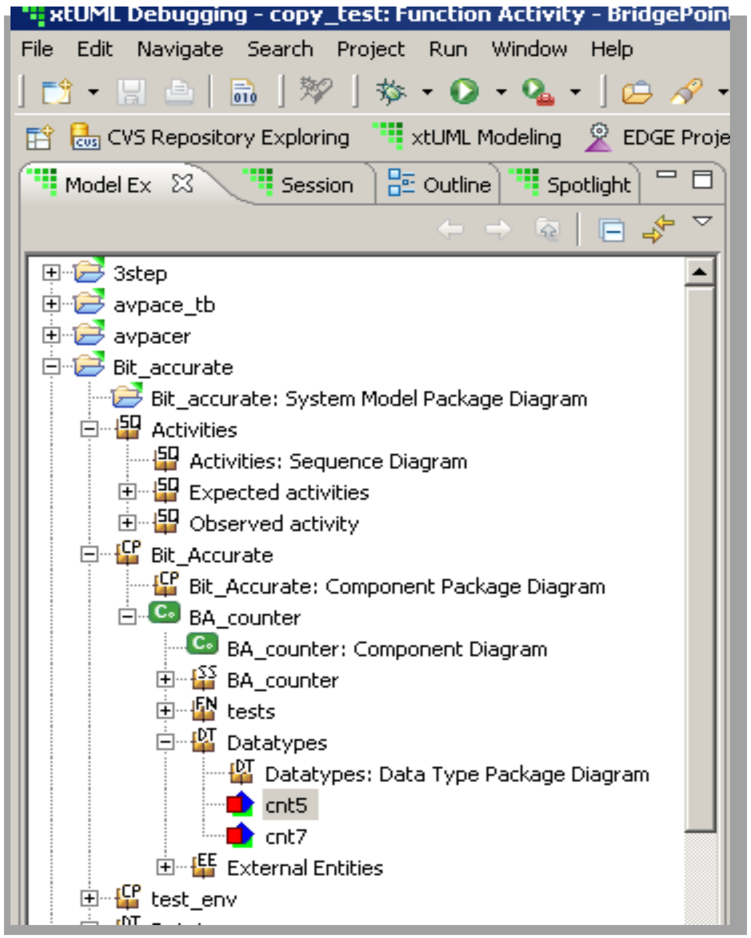
.invoke TagDataTypePrecision("domain", "dt_name", "tagged_name", "initial_value")

Where the input parameters are:

domain - Registered domain name. Use "*" to indictate a <u>System Wide</u> data type (to be applied to all domains containing the user data type).
dt_name - Name of the data type as known in the application analysis.
tagged_name - Name of the data type as known in generated implementation code (e.g., the 'precision' of the data type). Note (2).
initial_value - An optional specification of the default value for the data type. Use "" for the architectural default (e.g., 0 for integer, 0.0 for real). Note (3).

```
.invoke TagDataTypePrecision( "*", "cnt5", "sc_int<5>", "0" )
.invoke TagDataTypePrecision( "*", "cnt7", "sc_int<7>", "0" )
```

Generated Model




```
/*
 * Structural representation of application
      analysis class:
 *   counter  (COUNTER)
 */
class BA_counter_COUNTER {
  public:
  Escher_StateNumber_t current_state;
  /* application analysis class attributes */
  sc_int<5> counter;  /* - counter */
  /* - i_cnt7 */  /* OPTIMIZED OUT */
  sc_int<5> step_size;  /* - step_size */
```