Integer Data Type Semantics: SystemC™ vs. VHDL

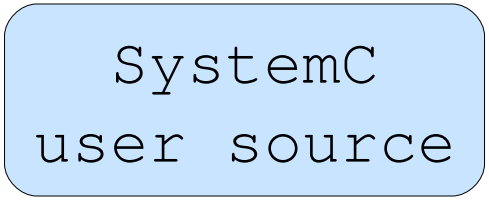
C. Brunzema (OFFIS e.V.)
F. Oppenheimer (OFFIS e.V.)

2008-09-23 FDL Stuttgart

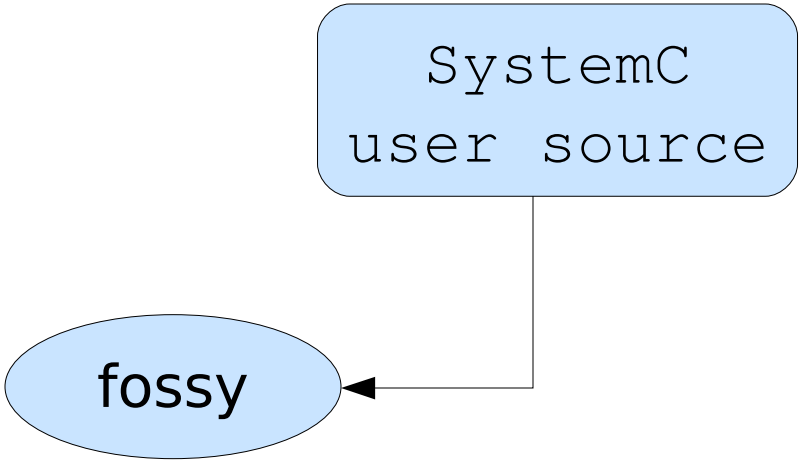
Overall Goal: SystemC Synthesis

- **Synthesis Flow**
- **Quirks of the SystemC integer data type semantics**
- **Quirks of the VHDL integer data types**
- **A simplified intermediate representation**
- **From the intermediate representation to VHDL**
- **Example Code**
- **Conclusion / Outlook**

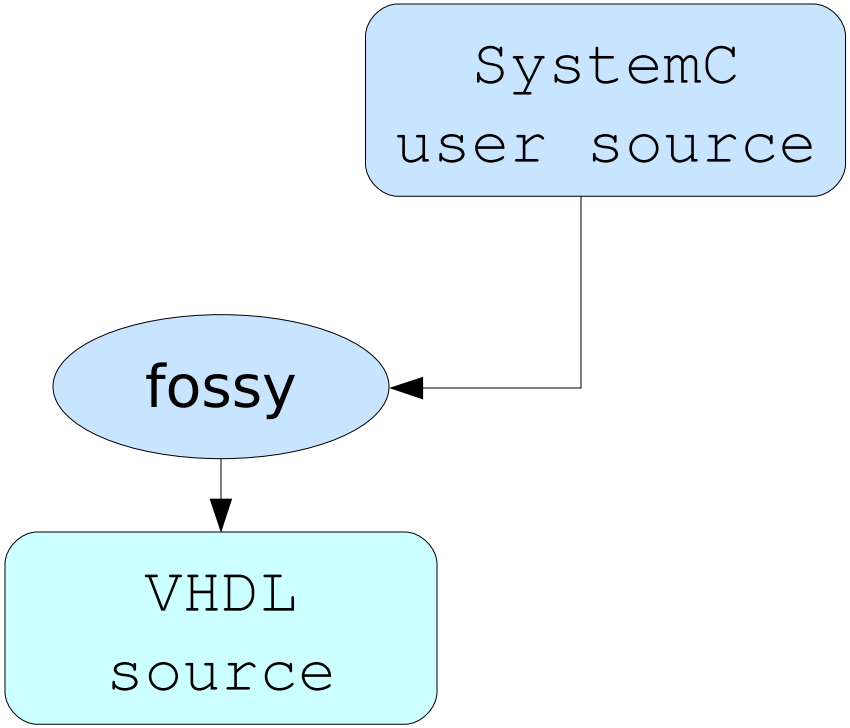
Synthesis Flow



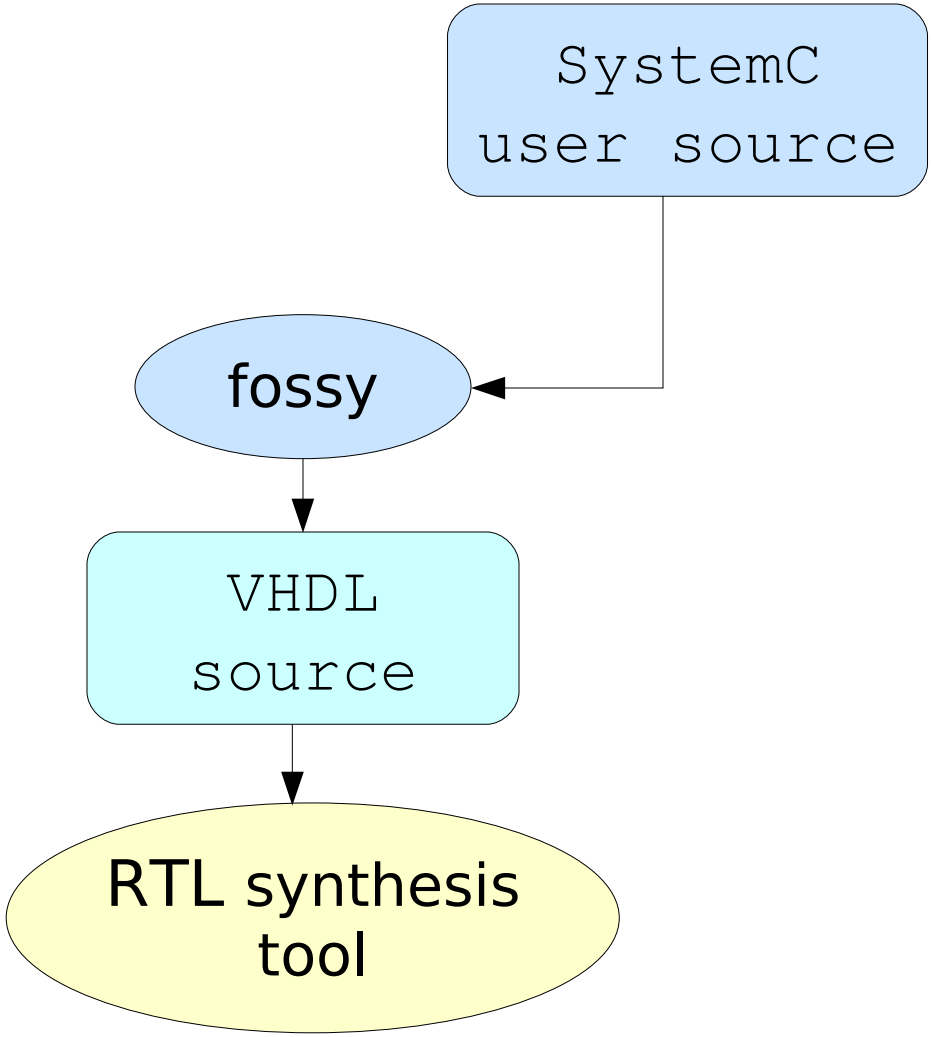
Synthesis Flow



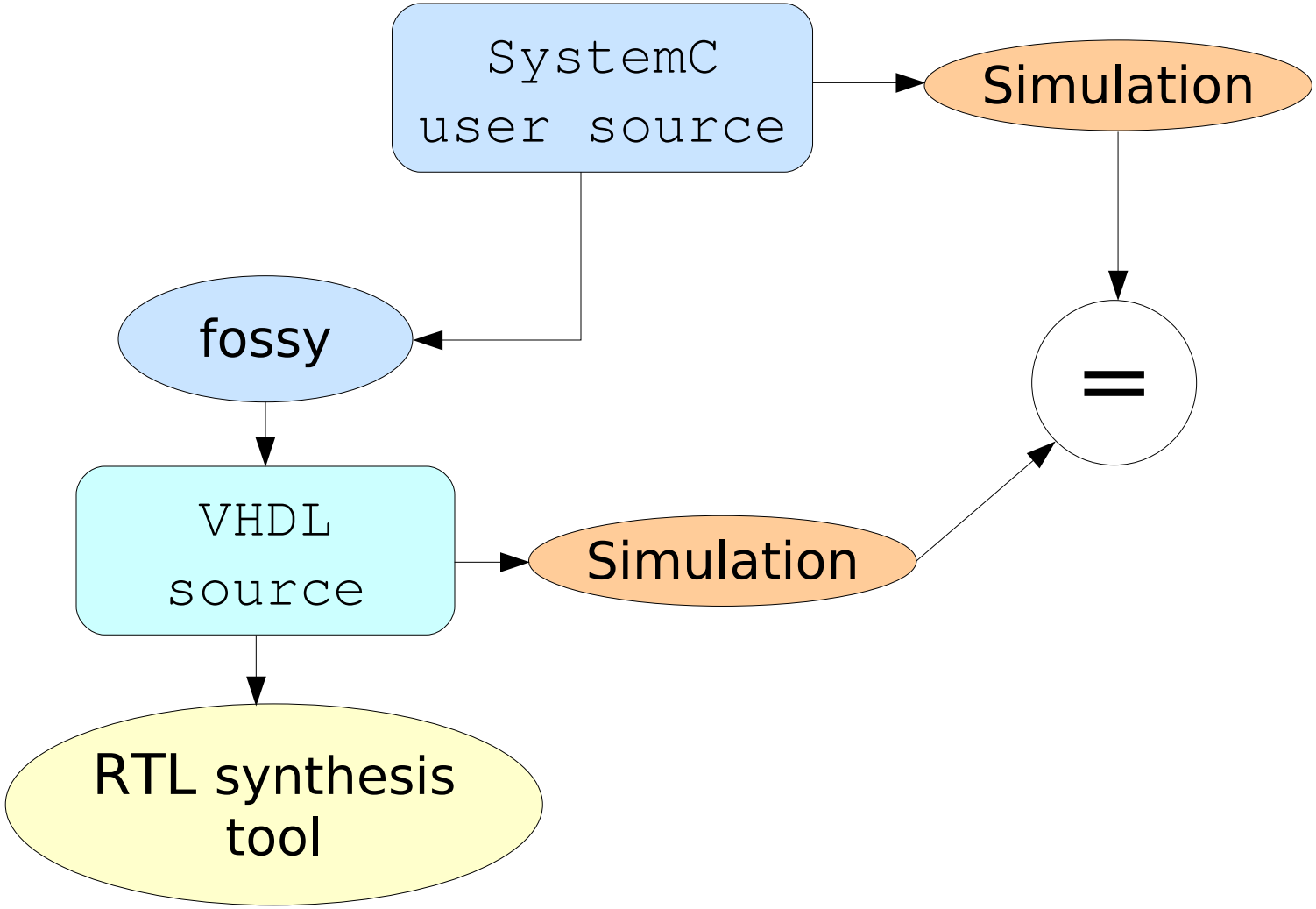
Synthesis Flow



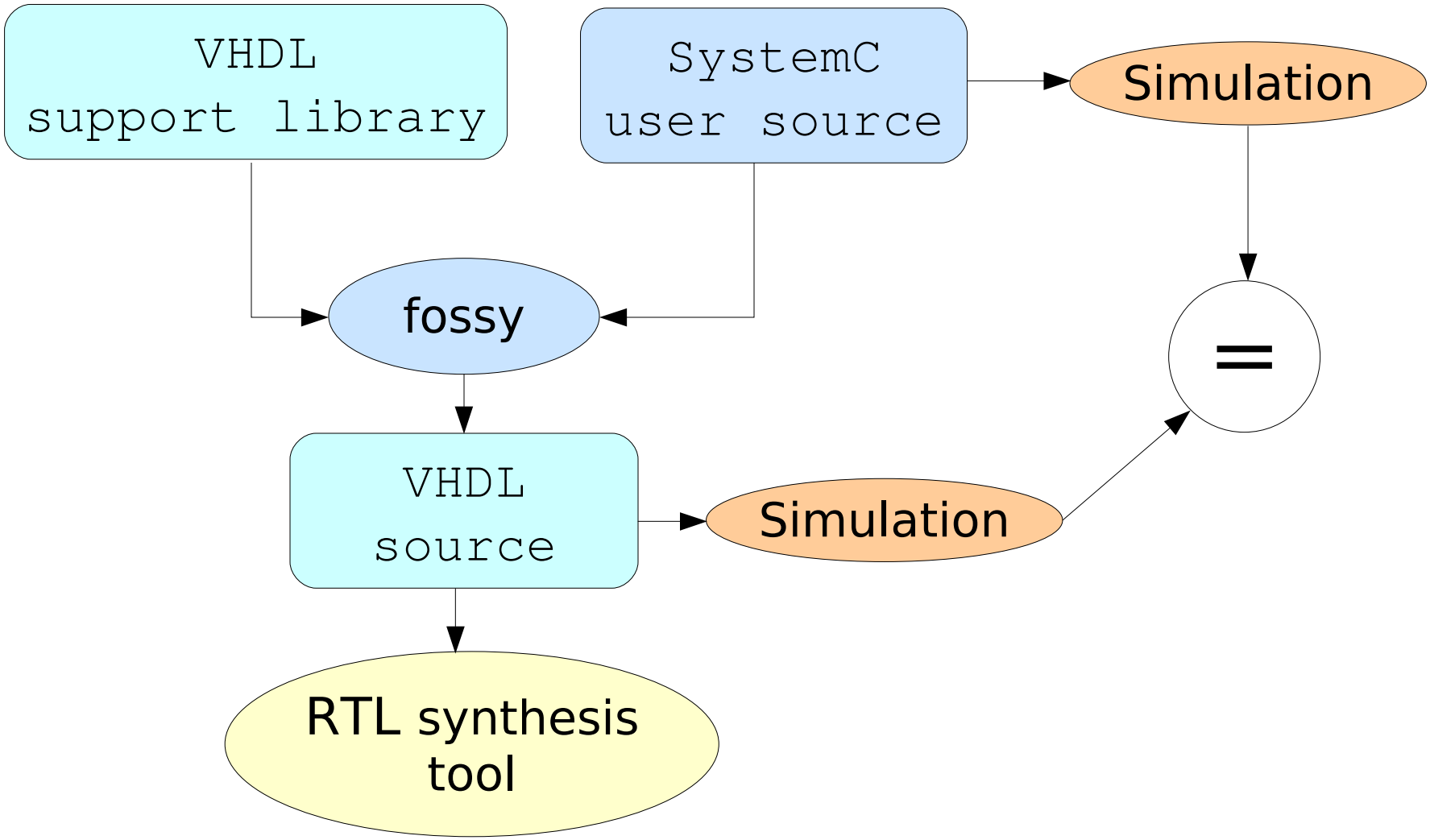
Synthesis Flow



Synthesis Flow



Synthesis Flow



SystemC Quirks (1)

sc_(u)int<W> ↔ sc_big(u)int<W>

sc_(u)int<W>

- **Must use native C++ data types internally**
- **Must support at least 64 bits**
- **Maximum bitwidth is implementation-dependent**
- **Allow high simulation speed due to the native internal representation**

sc_big(u)int<W>

- **Must support any bitwidth**
- **Flexibility has higher overhead and decreases simulation speed**

→ **Why not template specialisations on the bitwidth ?**

→ **The two distinct types have subtle differences**

→ **The two distinct types are not binding compatible**

→ **Lots of almost identical code**

SystemC Quirks (2)
Arithmetic

- **Arithmetic performed using hidden base classes without visible bitwidths**
  - sc_int<W>, sc_uint<W> → sc_int_base, sc_uint_base
  - sc_bigint<W>, sc_biguint<W> → sc_signed, sc_unsigned

- **... but invisible automatic sign extensions may cause overflow**

```
sc_uint<64> x = 18446744073709551615ul;    // max uint_64
        x + x → 18446744073709551614
sc_bigint<70>(x) + sc_bigint<70>(x) → 36893488147419103230
```

- **Result bitwidth of (unsigned – unsigned) is bigger than expected**
  - Subtraction always promotes its operands to signed

- **SystemC standard (p.214/p.220) is strange about result bitwidth of subtraction**
  - OSCI reference kernel apparently does the right thing

SystemC Quirks (2)
Arithmetic

- **Arithmetic performed using hidden base classes without visible bitwidths**
  - sc_int<W>, sc_uint<W> → sc_int_base, sc_uint_base
  - sc_bigint<W>, sc_biguint<W> → sc_signed, sc_unsigned

- **... but invisible automatic sign extensions may cause overflow**

```
sc_uint<64> x = 18446744073709551615ul;     // max uint_64
        x + x → 18446744073709551614        // overflow
sc_bigint<70>(x) + sc_bigint<70>(x) → 36893488147419103230
```

- **Result bitwidth of (unsigned – unsigned) is bigger than expected**
  - Subtraction always promotes its operands to signed

- **SystemC standard (p.214/p.220) is strange about result bitwidth of subtraction**
  - OSCI reference kernel apparently does the right thing

SystemC Quirks (3)

- **Division needs an extra bit to catch MAX_NEG / -1**
- **Extension of complements of unsigned is rather unintuitive**
  - maybe a bug in the OSCI kernel
  - `sc_biguint<8>(~ sc_biguint<3>("0b111")) → 0b011111000`
  - `sc_biguint<8>(~ sc_bigint<3>("0b111")) → 0b000000000`
- **Unbounded operator<< (shift left) on signed integers (p. 215)**
  - `(sc_bigint<4>(1) << 5).length() → 9`
  - `(sc_bigint<4>(1) << 50).length() → 54`
- **Ranges and bitselects are implemented with 20 helper classes:**

sc_int_bitref_r	sc_uint_bitref	sc_signed_subref_r	sc_unsigned_subref
sc_int_bitref	sc_uint_subref_r	sc_signed_subref	sc_bitref_r<>
sc_int_subref_r	sc_uint_subref	sc_unsigned_bitref_r	sc_bitref<>
sc_int_subref	sc_signed_bitref_r	sc_unsigend_bitref	sc_subref_r<>
sc_uint_bitref_r	sc_signed_bitref	sc_unsigned_subref_r	sc_subref<>

VHDL Quirks

- **Division needs an extra bit to catch MAX_NEG / -1**
  - IEEE 1076.3-1997 A2.1
- **Addition/Subtraction has no extra bit for carry/borrow**
  - This allows expressions like `A := A + 1;`
- **Range values are not expressions and retain the indices of their origin**
  - This is illegal: `(A(10 downto 5))(3 downto 0)`
- **Range values retain their signedness**
  - This is illegal: `unsigned_var(5 downto 3):= signed_var(2 downto 0);`

SystemC ↔ VHDL Comparison

SystemC	VHDL
Division needs special care	**Division needs special care**
Automatic argument expansion in arithmetic	**No automatic expansion of arguments**
Result bitwidths big enough for result values	**Result bitwidths of add/sub unchanged**
Unbounded shift left	**Sane shift left**
Ranges implemented with lots of classes	**Ranges may have signs and unusual indices, ranges are not expressions**

A simplified intermediate representation

- **Three basic data types: SIGNED, UNSIGNED, BITVECTOR**
  - sc_int<W>, sc_bigint<W> → SIGNED
  - sc_uint<W>, sc_biguint<W> → UNSIGNED
  - sc_bv<W> → BITVECTOR
- **All values have bit indices from MSB:(width-1) downto LSB:0**
- **All range expressions have type BITVECTOR**
- **Size changes (implicit/explicit casts, arithmetic extensions) become explicit RESIZE expressions**
- **Results of arithmetic operations have types/widths following SystemC semantics**
  - promote to signed when there is a signed type involved anywhere
  - use a types wide enough to hold the results
  - shift-left needs an explicit result bitwidth

Intermediate Representation → VHDL

- **Straightforward data type mapping:**

```
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
```

  SIGNED → SIGNED
  UNSIGNED → UNSIGNED
  BITVECTOR → STD_LOGIC_VECTOR

- **Results of range expressions need adjustment to (width-1) downto 0**
- **Results of range expressions need sign adjustment**
- **Casts/arithmetic extensions become calls to resize function**
- **SystemC semantics implemented in library using standard VHDL**
- **Use of `function`s results in prefix notation**

Example Code

```
sc_int<64> result;
sc_uint<5> x1 =   “0b11111”;       // 31
sc_uint<6> x2 = “0b111111”;       // 63

result = (x1 + x2) * 4;                  // 376
```

```
variable result : SIGNED(63 downto 0);
variable x1 : UNSIGNED(4 downto 0) := “11111”;
variable x2 : UNSIGNED(5 downto 0) := “111111”;

result := RESIZE( SIGNED("0" & (x1 + x2)) * TO_SIGNED(4, 32),
                        64);
--  result = 120

result := FOSSY_RESIZE(
                        FOSSY_MUL(
                                FOSSY_ADD(x1, x2),
                                TO_SIGNED(4, 32)),
                        64);
-- result = 376
```

Conclusion / Outlook

- **Our library helps converting SystemC arithmetic to VHDL**
- **Sign promotion and bitwidth details are hidden**
- **Standard VHDL types allow smooth integration**

- **We need to support more operations**
- **More syntactic sugar would be nice (operator overloading)**

Thank You

Free Software Download:

www.system-synthesis.org/download