System Design with ForSyDe

Ingo Sander

Royal Institute of Technology
Stockholm, Sweden
ingo@kth.se

January 12, 2011

Outline

1 Background
2 Haskell
3 ForSyDe Modeling Concepts
4 Design Transformation
5 Shallow Embedded ForSyDe - Modelling Heterogeneous Systems
6 Modeling of Adaptive Systems
7 Deep Embedded ForSyDe - Hardware Synthesis, Graphical Output
8 The SYSMODEL Project
9 Conclusion and Future Work
Observations

- The design productivity gap is still increasing
- Share of the verification costs of the total design costs is increasing
- Industry aims for more abstract design languages, but
  - SystemC and Verilog are further development of old language paradigms
  - Simulation is still the predominant verification technique
  - Difficult to apply formal methods since today’s design languages do not offer a sufficient formal basis

Formal and systematic approach is needed!

What is ForSyDe?

ForSyDe (Formal System Design) is a design methodology for systems-on-chip that has been developed with the following objectives.

- System design must start at a high level of abstraction
  - Designer shall focus on functionality
  - Low-level implementation details shall not be an issue at this stage
- Design methodology must give a solid base for the incorporation of formal methods
  - Verification must be a first class citizen from the start
- Abstraction gap between specification and implementation must be bridged by formal refinement technique
- ForSyDe allows to describe heterogeneous models at different abstraction levels.
- The idea is to 'map' a refined ForSyDe model to an implementation language such as VHDL or C.

ForSyDe Design Flow
Current Status of ForSyDe

- ForSyDe is implemented as domain specific language in Haskell.
- Several libraries for different models of computation exist and can be simulated as integrated model.
- ForSyDe processes are formally defined.
- ForSyDe supports modeling at different levels of abstraction.
- There exists a back-end for hardware design and synthesis (VHDL).
- High-level and synthesizable models can be co-simulated giving access to powerful test benches.

The Functional Language Haskell

- A functional program is a function that receives the program’s input as argument and delivers the program’s output as result.
- Haskell is a pure and lazy functional language
  - Pure Haskell is free from side-effects, i.e. there is no global state
  - Lazy Haskell is a lazy language, which means that the arguments will first be evaluated, when they are needed
- A functional program is a function that consists of other functions.
- A functional program contains implicit parallelism.

\[ r(x, y) = h(f(x), g(y)) \]

Here there are no data dependencies between \( f(x) \) and \( g(y) \). Thus they can be executed in any order or in parallel.
Length of a List

The length of the list can be defined recursively.

There are two cases:

1. The list is empty
2. The list has at least one element

This can be directly modelled in Haskell:

```haskell
len [] = 0  -- (1) List is empty
len (hd:tl) = 1 + len tl  -- (2) List is non-empty
```

Execution in the Haskell interpreter, here `ghci`, gives

```haskell
*HaskellIntro> len [7,2,5,4,9,3,2]
7
*HaskellIntro> len ['a','b','c']
3
```

Program Execution

The program

```haskell
len [] = 0  -- (1) List is empty
len (hd:tl) = 1 + len tl  -- (2) List is non-empty
```

is executed in the following steps:

```haskell
len ['a','b','c']
= len ('a' : ['b', 'c'])  -- Pattern 2 matches
= 1 + len ['b', 'c']
= 1 + len ('b': ['c'])  -- Pattern 2 matches
= 2 + len ['c']
= 2 + len ('c' : [] )  -- Pattern 2 matches
= 3 + len []  -- Pattern 1 matches
= 3 + 0
= 3
```
Quicksort

Complex algorithms can be formulated very concise and clearly!

```haskell
quicksort [] = []
quicksort (x:xs)
  = quicksort [y | y <- xs, y <= x]
    ++ [x]
    ++ quicksort [y | y <- xs, y > x]
```

*HaskellIntro> quicksort [7,2,5,4,9,3,2]
[2,2,3,4,5,7,9]

Higher-order function: map

A higher-order function is a function that takes functions as argument and/or produces a function as output.

```haskell
map f [] = []   -- empty list
map f (x:xs) = f x : map f xs -- all other lists
```

The higher-order function map can be used with different data types.

*HaskellIntro> intlist
[7,2,5,4,9,3,2]
*HaskellIntro> map even intlist
[False,True,False,True,False,False,True]
*HaskellIntro> map (*3) intlist
[21,6,15,12,27,9,6]
Haskell is strong-typed!

map \( f \) \([\,]\) = \([\,]\) \hspace{1cm} \text{-- empty list}

map \( f \) \((x:xs)\) = \( f \) \( x \) : map \( f \) \( xs \) \hspace{1cm} \text{-- all other lists}

The Haskell type system can infer the data type of a function.

\*HaskellIntro* \> :i map

map :: \((a \to b) \to [a] \to [b]\)

This means that the function map takes a function that takes two arguments:
1. a function that takes a data type \( a \) and returns a data type \( b \)
2. a list of data type \( a \)

The function map returns a list of data type \( b \).

Wrong types are rejected

map \( f \) \([\,]\) = \([\,]\) \hspace{1cm} \text{-- empty list}

map \( f \) \((x:xs)\) = \( f \) \( x \) : map \( f \) \( xs \) \hspace{1cm} \text{-- all other lists}

Since Haskell expects a list as its second argument map `even 'c'` is rejected!

\*HaskellIntro* \> map even 'c'

<interactive>:1:9:

    Couldn’t match expected type ‘[a]’ against inferred type ‘Char’
    In the second argument of ‘map’, namely ‘c’
    In the expression: map even ‘c’
    In the definition of ‘it’: it = map even ‘c’
Function Composition

The operator \( \circ \) allows function composition.

\[
(f \circ g)x = f(g(x))
\]

In Haskell ‘.’ is used instead for \( \circ \).

```haskell
inc x = x + 1
newFunction = even . inc

*HaskellIntro> map even [1,2,3,4]
[False, True, False, True]
*HaskellIntro> map newFunction [1,2,3,4]
[True, False, True, False]
```

Function Application Operator

The operator \( $ \) is used for function application.

\[
(\$) : : (a \rightarrow b) \rightarrow a \rightarrow b
f \$ x = f \ x
\]
ForSyDe System Model

- A system is modelled as hierarchical concurrent process model
- Processes of different models of computation communicate via domain interfaces
  - Supported MoCs: Synchronous, Untimed (Synchronous Data Flow), Continuous Time

![Diagram of ForSyDe System Model]

ForSyDe Process

A process takes \( m \) input signals as argument and produces \( n \) output signals. ForSyDe processes are deterministic.

- A process is always designed by means of a **process constructor**
- The process constructor defines the communication interface of the process
- The process constructor takes side-effect free **functions** and **variables** as arguments and returns a process

\[
\text{Process Constructor} + \text{Functions} + \text{Variables} = \text{Process}
\]
Process Constructor Types

There are three main categories of process constructor

Combinational  Process has no internal state
  Delay  Process delays input
Sequential  Processes have an internal state and contain a delay process

These categories exist in all models of computation

Synchronous Process

Communication and Synchronization

Process

Signal

Synchronous Assumption

The outputs of a system (process) are synchronized with the inputs. The reaction of the system is instantaneous and takes no observable time!
Process Constructor - Benefits

The concept of process constructor

- separates communication from computation
  - process constructor: communication and interface
  - function: computation
- forces the designer to develop a structured formal model that allows for formal analysis
  - transformational refinement
  - implementation mapping
  - formal verification

Semantic-Preserving Design Transformations

Semantic-Preserving Transformation

\( \text{mapSY}(+2) \circ \text{mapSY}(+5) \) can be transformed into \( \text{mapSY}(+7) \)

Possible in the synchronous model, since computation takes no time (not even a delta delay)!
Non-Semantic-Preserving Design Transformations

Also non-semantic preserving design transformations are needed in order to arrive at an efficient implementation
- requires verification, since semantics are changed

Concepts and verification method have been developed inside the ForSyDe project

Modelling Heterogeneous Systems

The Shallow Embedded Implementation of ForSyDe

ForSyDe has initially been mainly developed for the modelling purpose.

- Signals have been modelled as streams of data.
  
  data Signal a = NullS | a :- Signal a

- Pros
  - Rapid modelling of heterogeneous systems
  - No need for advanced or non-standard features of Haskell
  - Easy to include new models of computation

- Cons
  - Restriction to simulation only, since there is no access to the abstract syntax tree
Modeling combinational processes

\[ \text{mapSY} \]  

\[
\text{mapSY} :: (a \to b) \to \text{Signal} \ a \to \text{Signal} \ b  \\
\text{mapSY} \ _ \ _ \ \text{NullS} = \text{NullS}  \\
\text{mapSY} \ f \ (x:.-xs) = f \ x \ :- \ (\text{mapSY} \ f \ xs)
\]

Example

\text{inverter} = \text{mapSY} \ \text{not}

\[
\text{HaskellIntro} > \text{inverter} \ (\text{signal} \ [\text{True}, \text{False}]) \ \\
\{\text{False}, \text{True}\}
\]

Modeling combinational processes

\[ \text{zipWithSY} \]  

\[
\text{zipWithSY} :: (a \to b \to c) \to \text{Signal} \ a \to \text{Signal} \ b \to \text{Signal} \ c  \\
\text{zipWithSY} \ _ \ _ \ _ \ \text{NullS} = \text{NullS}  \\
\text{zipWithSY} \ _ \ _ \ _ \ \text{NullS} = \text{NullS}  \\
\text{zipWithSY} \ f \ (x:.-xs) \ (y:.-ys)  \\
\quad = f \ x \ y \ :- \ (\text{zipWithSY} \ f \ xs \ ys)
\]

Example

\text{adder} = \text{zipWithSY} \ (+)
Modeling sequential processes

\[ i \xrightarrow{delaySY_k (s_0)} o \]

The process constructor `delaySY`

\[
delaySY :: a \rightarrow Signal a \rightarrow Signal a
delaySY e es = e:es
\]

Example

\[
\text{register} = \text{delaySY } 0
\]

Modeling sequential processes

\[ i_1 \rightarrow \text{scanldSY}_m(f, s_0) \rightarrow \text{zipWithSY}_{m+1}(f) \rightarrow s \rightarrow \text{delaySY}_1 (s_0) \rightarrow o \]

The process constructor `scanldSY`

\[
\text{scanldSY} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow Signal b \rightarrow Signal a
\]

\[
\text{scanldSY } f \text{ mem } \text{ xs} = s'
\]

where \( s' = \text{delaySY } \text{ mem } (\text{zipWithSY } f s' \text{ xs}) \)

Example

\[
\text{counter} = \text{scanldSY } (+) 0
\]
Modeling sequential processes

\[
\text{mooreSY}_m(f, g, s_0)
\]

\[
\text{scanlDSY}_m(f, s_0) \rightarrow s \rightarrow \text{mapSY}(g)
\]

The process constructor \text{mooreSY}

\[
\text{mooreSY} :: (a \rightarrow b \rightarrow a) \rightarrow (a \rightarrow c) \\
\rightarrow a \rightarrow \text{Signal b} \rightarrow \text{Signal c}
\]

\[
\text{mooreSY} \text{ nextState output initial} = \text{mapSY output} . (\text{scanlDSY} \text{ nextState initial})
\]

Tutorial Example
Simulating a Heterogeneous System

1. Synchronous Input Signal
   \[ \text{in} = \{0,1,2,3,4,5\} \]

3. Transceiver Output

4. Gaussian Noise

5. Noisy Transceiver Input

7. Synchronous Output Signal
   \[ \text{out} = \{0,1,10,3,4,5\} \]

Simplified DES Algorithm

- Both the encryption and decryption units are implemented with the simplified DES algorithm
- The simplified DES algorithm has similar properties and structure as DES, but is developed for educational purposes.

Simplified DES Algorithm

\[
\begin{align*}
\text{ciphertext} & = (\text{IP}^{-1} \circ f_{K_2} \circ \text{SW} \circ f_{K_1} \circ \text{IP}) \text{ plaintext} \\
\text{plaintext} & = (\text{IP}^{-1} \circ f_{K_1} \circ \text{SW} \circ f_{K_2} \circ \text{IP}) \text{ ciphertext}
\end{align*}
\]

- \(\text{IP}\): initial permutation
- \(\text{IP}^{-1}\): inverse of initial permutation
- \(\text{SW}\): switches halves of data
- \(f_{K_i}\): permutation and substitution of data using the subkey \(i\). Two 8-bit subkeys are generated from an initial 10-bit key.
Simplified DES - The $f_K$ Functionality

Process networks are modelled as set of equations.

```
import Data.Param.FSVec
import qualified Data.Param.FSVec as FS

f :: FSVec D8 Bit -> FSVec D4 Bit -> FSVec D4 Bit
f subkey nibble
    = p4 (out_S0 FS.++ out_S1)
    where out_S0 = s0matrix out_xor
          out_S1 = s1matrix out_xor
          out_xor = zipxor subkey out_ep
          out_ep = exp_perm nibble

f_k :: FSVec D8 Bit -> FSVec D8 Bit
    -> FSVec D8 Bit
f_k subkey input
    = outLeft FS.++ outRight
    where
          outLeft = FS.zipWith xor inpLeft fOut
          outRight = inpRight
          fOut = f subkey inpRight
          inpLeft = FS.take d4 input
          inpRight = FS.drop d4 input
```

**Fixed-Sized Vectors**

For hardware design vectors of fixed size are of crucial importance for both modelling and an efficient design implementation. ForSyDe offers the data type FSVec\(^1\) that allows to specify vectors of constant size, which can also be synthesized to hardware.

```
import Data.Param.FSVec
import qualified Data.Param.FSVec as FS

splitBlock :: FSVec D8 Bit -> (FSVec D4 Bit, FSVec D4 Bit)
splitBlock block = (FS.take d4 block, FS.drop d4 block)
```

Fixed Size Vectors are heavily used during the tutorial example.

\(^1\)The implementation was done by Alfonso Acosta
Adaptive Processes

- Basic Idea
  - Functions can be used as signal values
  - Adaptive process executes function that is provided at time instance for execution

\[
s_a = \{2, 4, 6, 8, \ldots\}
\]

\[
s_b = \{1, 2, 3, 4, \ldots\}
\]

Functionality is provided from the outside and "loaded" into the adaptive process.

Adaptivity in Haskell

Since functions are first-class citizens, adaptivity can be easily implemented in Haskell.

**Implementation of Function Adaptivity**

```haskell
funcAdaptSY = zipWithSY ($)
funcAdapt2SY = zipWith3SY ($)

s_f = signal ([(+),(x),(+),(x),\ldots])
```

```haskell
*HaskellIntro> s_a
{2,4,6,8}  
*HaskellIntro> s_b
{1,2,3,4}  
*HaskellIntro> funcAdapt2SY s_f s_a s_b
{3,8,9,32}
```
An illustrative example

Different levels of adaptivity

Adaptive processes can be of different complexity

**Parameter**  Adaptive process contains a parameter that can be changed from the outside

**Mode**  Adaptive process contains several functions that can be selected from the outside

**Function**  Adaptive process changes its functionality depending on the function that is supplied from the outside

**Interface**  Adaptive process changes its functionality and interface depending on the process that is supplied from the outside
- Number of inputs and outputs can change
- MoC can change
- Functionality can change
**Parameter Adaptivity**

Functionality is part of the process, but is modified by a parameter that is provided from the outside.

\[
s_a = <3, 4, 5, 3, \ldots>
\]

\[
s_o = <3, 8, 10, 3, \ldots>
\]

\[
s_p = <1, 2, 1, \ldots>
\]

Functionality of \(P_P\)

\[
s_o(i) = s_a(i) \cdot s_p(i)
\]

**Mode Adaptivity**

Functionality is part of process and controlled from the outside.

\[
s_a = <2, 4, 6, 8, \ldots>
\]

\[
s_b = <1, 2, 3, 4, \ldots>
\]

\[
s_m = <ADD, MUL, ADD, MUL, \ldots>
\]

Functionality of \(P_M\)

\[
s_o(i) = \begin{cases} 
  s_a(i) + s_b(i) & | \quad s_m(i) = ADD \\
  s_a(i) \cdot s_b(i) & | \quad s_m(i) = MUL
\end{cases}
\]
Function Adaptivity

Functionality is provided from the outside and "loaded" into the adaptive process.

\[ s_a = <2, 4, 6, 8, \ldots > \]
\[ s_b = <1, 2, 3, 4, \ldots > \]
\[ s_f = <(+), (x), (+), (x), \ldots > \]

Functionality of \( P_F \)
\[ s_o(i) = s_f(i)(s_a(i), s_b(i)) \]

Interface Adaptivity

Adaptive process changes its functionality and interface depending on the "loaded" process.

\[ s_i = <P_{SY,1}(i_1, i_2), P_{SDF,1}(i_m), P_{SY,2}(i_2), \ldots > \]
Interface Adaptivity

Adaptive process changes its functionality and interface depending on the "loaded" process

\[ s_i = < P_{SY,1}(i_m), P_{SY,2}(i_2), \cdots > \]
Interface Adaptivity

Interface Adaptive Process

\[ s_i = \langle P_{SY,1}(i_1, i_2), P_{SDF,1}(i_m), P_{SY,2}(i_2), \ldots \rangle \]

- Interface Adaptive Processes are inherently complex
  - many open questions!
- However, they have their counterpart in the implementation domain
- Any function can be loaded on the fly onto a partially reconfigurable FPGA (e.g. Xilinx Virtex IV-family)

Partially reconfigurable FPGAs

Just-In-Time Adaptivity

- Adaptive processes can be implemented using a run-time reconfigurable FPGA
  - Configuration (process/function) is loaded from configuration memory into configuration slot on FPGA
  - Non-reconfigurable area performs its function even during reconfiguration!
Design Refinement

- An efficient design flow starts at high abstraction level
  - functionality shall be analyzed
  - implementation details are not necessary at this stage

Adaptation is assumed to be instantaneous

Design Refinement

- During the design process implementation details are added
  - Adaptation is no longer instantaneous ⇒ Buffers have to be introduced
Design Refinement

- Design process for adaptive systems imposes hard requirements on correctness
  - System must work correctly during course of adaptation, which can comprise several stages
  - Performance data is required Difficult to ensure correctness by ad hoc refinement

Formal approach is needed!

Deep-Embedded ForSyDe

In order to allow to give structural information to a ForSyDe model, recently a deep-embedded version of ForSyDe has been developed. The main objectives have been

- to allow for efficient hardware synthesis of synchronous ForSyDe models
- to be able to simulate synchronous ForSyDe models
- to be able to integrate further back-ends for analysis and transformation
- to be able to combine and co-simulate shallow-embedded and deep-embedded ForSyDe models
Why an Embedded Compiler?

- **Feasibility.** Does require less time than development of traditional compiler
- **Saves unnecessary effort.** The goal is to translate the system structure, not any arbitrary Haskell program
- **Previous success.** Successful embedded compiler has been developed for Lava
- **Maintainable.** The compiler is packed with ForSyDe’s Library
- **Independent of third-party tools.** No risk of getting outdated due to external design changes

Process Function

- ForSyDe uses Template Haskell to be able to extract the Abstract Syntax Tree.
- Each function that is argument to a process constructor needs to be declared using Template Haskell.

```haskell
p4Fun :: ProcFun (FSVec D2 Bit -> FSVec D2 Bit -> FSVec D4 Bit)
p4Fun = $(newProcFun
    [d| p4 :: FSVec D2 Bit -> FSVec D2 Bit -> FSVec D4 Bit
    p4 outS0 outS1 = outS0!d1 -> outS1!d1 ->
    outS1!d0 -> outS0!d0 -> empty |])
```

Use of Template Haskell

- The declaration `[d| ... |]` is used to give a definition of a function so that its abstract syntax tree is accessible at compile-time.
- `$(...)` means ”evaluate at compile time”. Here a `newProcFun` using the declaration in `[d| ... |]` is evaluated at compile-time and gives access to the abstract syntax tree.
System Function and Definition

Processes are created using process constructors and process functions as arguments. An additional argument is the process identifier that can be used by the compiler back-end.

\[
p4Proc :: \text{Signal (FSVec D2 Bit)} \to \text{Signal (FSVec D2 Bit)} \\
\phantom{p4Proc} \to \text{Signal (FSVec D4 Bit)}
\]

\[
p4Proc = \text{zipWithSY "p4Proc" p4Fun}
\]

A system function describes a system. Additional parameters are system identifier and input and output ports.

\[
p4Sys :: \text{SysDef (Signal (FSVec D2 Bit)} \to \text{Signal (FSVec D2 Bit)} \\
\phantom{p4Sys} \to \text{Signal (FSVec D4 Bit))}
\]

\[
p4Sys = \text{newSysDef p4Proc "p4" ["S0", "S1"] ["out"]}
\]

Composing a Larger System

Systems can be used as components to create larger systems.

\[
fProc :: \text{Signal (FSVec D4 Bit)} \to \text{Signal (FSVec D8 Bit)} \\
\phantom{fProc} \to \text{Signal (FSVec D4 Bit)}
\]

\[
fProc \text{ nibble subkey = out}
\]

where

\[
\begin{align*}
\text{out} & = \text{(instantiate "p4Sys" p4Sys) s0 s1} \\
\text{s0} & = \text{(instantiate "outputS0Sys" outputS0Sys) rS0 cS0} \\
\text{s1} & = \text{(instantiate "outputS1Sys" outputS1Sys) rS1 cS1} \\
\text{rS0} & = \text{(instantiate "rowS0" rowS0Sys) xorSubkey} \\
\text{rS1} & = \text{(instantiate "rowS1" rowS1Sys) xorSubkey} \\
\text{cS0} & = \text{(instantiate "colS0" colS0Sys) xorSubkey} \\
\text{cS1} & = \text{(instantiate "colS1" colS1Sys) xorSubkey} \\
\text{xorSubkey} & = \text{(instantiate "xorSubkeySys" xorSubkeySys) subkey expperm} \\
\text{expperm} & = \text{(instantiate "expPermSys" expPermSys) nibble}
\end{align*}
\]

\[
fSys :: \text{SysDef (Signal (FSVec D4 Bit)} \to \text{Signal (FSVec D8 Bit)} \\
\phantom{fSys} \to \text{"Signal (FSVec D4 Bit))}
\]

\[
fSys = \text{newSysDef fProc "fSys" ["nibble", "subkey"] ["out"]}
\]
Simulation

Deep-embedded models can be simulated using the command `simulate`. The system process `encryptSys` encrypts a plain text and `decryptSys` is the corresponding decryption.

```plaintext
key1 = [H -> L -> H -> L -> L -> L -> L -> L -> H -> L -> empty]
subkey_1 = fst $ simulate subkeysSys key1
subkey_2 = snd $ simulate subkeysSys key1
enc = simulate encryptSys subkey_1 subkey_2 plain
dec = simulate decryptSys subkey_1 subkey_2 enc

> plain
> enc
> dec
```

Deep-embedded and shallow-embedded models can be co-simulated in the 'shallow-embedded world'.

```plaintext
combSim shallowSig = (shallowProcess . signal . simulate DeepSys . fromSignal) shallowSig
```

Graphical Output

It is possible to obtain a graphical representation using ForSyDe's GraphML back-end.

```plaintext
> writeGraphMLOps defaultGraphMLOps{yFilesMarkup=True} fSys
```

The diagram can then be viewed using the editor `yEd` from `yWorks`:
Hardware Synthesis - Processes

The main ideas for hardware synthesis are summarized in this and the following slide:

- Each process constructor has a corresponding VHDL-template
- ForSyDe functions and data types are translated to VHDL

```
mapSY f               fHW
\downarrow
delaySY v             Register
\downarrow
\downarrow
\uparrow
mooreSY f, g, v       fHW
\downarrow
\downarrow
\downarrow
\uparrow
```

Hardware Synthesis - Process Networks

ForSyDe process networks are mapped to networks of components

- ForSyDe signals are mapped to VHDL signals
- ForSyDe processes are mapped to VHDL components
VHDL Synthesis

ForSyDe’s embedded compiler is able to translate system definitions to VHDL. This is done through the `writeVHDL` function.

```
> writeVHDL fSys
```

The function generates a tree structure containing the VHDL code of all included subsystems.

FPGA Synthesis using Altera Quartus

ForSyDe allows to directly create a downloadable file for a specific Altera FPGA circuit, given that the Altera tools are installed.

Synthesis of encryptSys

```
compileQuartus_encryptSys :: IO ()
compileQuartus_encryptSys = writeVHDLOps vhd10ps encryptSys
  where vhd10ps = defaultVHDLOps{execQuartus=Just quartusOps}
  quartusOps = QuartusOps{action=FullCompilation,
                           fMax=Just 50, -- in MHz
                           fpgaFamilyDevice=Just ("CycloneII",
                                                  Just "EP2C35F672C6"),
                           -- Possibility for Pin Assignments
                           pinAssigs=[]
}
```

Quartus generates then reports, which inform about size and speed of the design (encryptSys needs 36 logic elements, 32 pins, and the worst case delay is 18.836 ns)
FPGA Synthesis Using Altera Quartus

Output from RTL Viewer

• Implementation of $\text{encryptSys}$

• Implementation of $\text{fkSys}$
The SYSMODEL Project

- The Artemis project 'SYSMODEL' (System Level Modeling Environment for SMEs) addresses the design process of heterogeneous embedded systems.

- Industrial Partners: Technoconsult, SIB Development (Denmark), Sting Networks, Catena, Solidux (Sweden), DA-Design, Finnelpro (Finland), Novelda (Norway)

- Academic Partners: Technical University of Denmark, Royal Institute of Technology (Sweden), Tampere University of Technology (Finland)

- The project has started in January 2009 and has a duration of three years.

SYSMODEL: Objectives

The SYSMODEL project follows the platform-based design approach. There are the following main objectives:

- Development of modeling and simulation framework for analysis and design of embedded systems and systems-on-chip.
  - System Functionality Framework
  - Platform Architecture Framework
  - Design Space Exploration: Analysis, Mapping and Testability
  - Verification of models

The results of the project shall increase productivity in industry.
Models and Languages

- To provide a base for formal methods, the system functionality framework is based on the ForSyDe modeling framework.
- To allow for industrial exploitation SystemC is used as the main modeling language both for the system functionality framework and the platform architecture framework.
  - Already today several dialects for different models of computation exist. However, they often lack a formal semantics.
  - Other languages, such as VHDL for hardware models or C/C++ for algorithms shall be integrated in the framework.
- An industrial 'refinement-by-replacement' approach shall be supported. Parts of the system functionality models shall be replaced by executable platform models and then co-simulated with the system functionality models.

ForSyDe System Model

- A system is modeled as hierarchical concurrent process model
- Processes of different models of computation communicate via domain interfaces
  - Supported MoCs: Synchronous, Untimed (Synchronous Data Flow), Continuous Time

Systems containing analog, digital and software parts can be modeled at different levels of abstraction!
SystemC

- SystemC is a class library built on top of the C++ language.
- SystemC adds concurrency and notion of time to C++ to allow to model systems.
- The semantics of SystemC is very different from C++.
- Systems in SystemC are modeled network of communicating processes. Computational processes are encapsulated in modules, while the communication among them is performed through channels.

SYSMODEL: ForSyDe is implemented in SystemC

SYSMODEL: Modeling rules will be developed in order to write SystemC models to comply to the ForSyDe semantics!
The System Functionality Model is modeled in SystemC. Other languages can be imported by means of SystemC-wrappers. The whole model can be co-simulated.

A SystemC block has been refined and replaced by C-code that runs on an instruction set simulator, which belongs to the platform architecture framework. The whole model can be co-simulated.
The ANDRES project

ForSyDe has been used as formal framework in the FP6 ANDRES project: (Analysis and Design of run-time Reconfigurable, heterogeneous Systems). The ANDRES project used SystemC as design language. As part of the project:

• a specification methodology
• a performance analysis method

for reconfigurable systems has been developed.

Facts on ANDRES

• Industrial Partners: Thales Communications (France), DS2 (Spain)
• Academic Partners: OFFIS (Germany), Royal Institute of Technology (Sweden), Technical University Vienna (Austria), University of Cantabria (Spain)
• The project has started in June 2006 and will finish in September 2009.

Conclusion

• Complex adaptive heterogeneous systems can be modelled and simulated in ForSyDe
• Deep-embedded ForSyDe implementation allows to extract structural information of a ForSyDe model
  • used for hardware synthesis
    • Connection to Altera Quartus (FPGA-Synthesis)
    • Connection to Modelsim (VHDL-Simulator)
  • used for graphical output (GraphML)
• So far deep-embedded presentation is only possible for synchronous models
• Not all desired constructs can yet be synthesized
• Template Haskell allows to extract the structural information, but implies a less intuitive syntax
Future Work

- Extension of VHDL-backend
  - More synthesizable constructs
    - At present only Int8, Int16, and Int32 are supported
    - Higher-order functions on fixed-sized vectors cannot be synthesized, such as FS.zipWith.xor s1 s2.
  - Better support for pattern matching
  - Introduce optimization capabilities into VHDL-backend
    - So far compiler does only map ForSyDe-model to VHDL

- Extension of deep-embedded implementation to additional models of computation
  - So far only synchronous model is supported

- Adaptation to new problem areas
  - Software synthesis
  - Formal verification
  - Performance analysis
  - Design transformation

Thanks for your attention!

More information on Haskell
http://www.haskell.org/

More information on ForSyDe
http://www.ict.kth.se/forsyde/