ForSyDe

Rising the abstraction level in System Design

Alfonso Acosta
alfonsoa@kth.se

SAM/ECS/ICT/KTH
Royal Institute of Technology, Stockholm

March 27th, 2008

Outline

1. Introduction to ForSyDe
   - The abstraction gap problem
   - What is ForSyDe?
   - Examples
   - Simplified Design Flow
   - Initial implementation of ForSyDe

2. Current ForSyDe's implementation details
   - The compilation problem
   - Deep Embedding + Embedded compiler
   - The Sharing Problem: Observable Sharing
   - The Polymorphism Problem: Dynamic Types
   - Deep-embedding process-constructor parameters
   - Components

Outline

1. Introduction to ForSyDe
   - The abstraction gap problem
   - What is ForSyDe?
   - Examples
   - Simplified Design Flow
   - Initial implementation of ForSyDe
2. Current ForSyDe's implementation details
   - The compilation problem
   - Deep Embedding + Embedded compiler
   - The Sharing Problem: Observable Sharing
   - The Polymorphism Problem: Dynamic Types
   - Deep-embedding process-constructor parameters
   - Components

Outline

1. Introduction to ForSyDe
   - The abstraction gap problem
   - What is ForSyDe?
   - Examples
   - Simplified Design Flow
   - Initial implementation of ForSyDe

2. Current ForSyDe's implementation details
   - The compilation problem
   - Deep Embedding + Embedded compiler
   - The Sharing Problem: Observable Sharing
   - The Polymorphism Problem: Dynamic Types
   - Deep-embedding process-constructor parameters
   - Components

Outline

1. Introduction to ForSyDe
   - The abstraction gap problem
   - What is ForSyDe?
   - Examples
   - Simplified Design Flow
   - Initial implementation of ForSyDe
2. Current ForSyDe's implementation details
   - The compilation problem
   - Deep Embedding + Embedded compiler
   - The Sharing Problem: Observable Sharing
   - The Polymorphism Problem: Dynamic Types
   - Deep-embedding process-constructor parameters
   - Components

The abstraction-gap problem

- Systems designed nowadays (and electronic components in particular) are increasingly complex.
- Market pressure calls for two appartenly opposite constraints:
  1. Efficiency $\Rightarrow$ It is required to handle low-level details at design time $\Rightarrow$ **low abstraction level**
     - *Example: Large layout parts of today's high performance processors are designed and optimized by hand.*
  2. Low time-to-market and complex features $\Rightarrow$ It is preferable to avoid dealing with low-level details $\Rightarrow$ **high abstraction level**
     - *Example: SoC (System on Chip) architectures embed multiple heterogeneous components*
- ForSyDe's main motivation is to solve the resulting abstraction gap problem

The abstraction-gap problem

- Systems designed nowadays (and electronic components in particular) are increasingly complex.
- Market pressure calls for two appartenly opposite constraints:
  1. Efficiency $\Rightarrow$ It is required to handle low-level details at design time $\Rightarrow$ **low abstraction level**
     - *Example: Large layout parts of today's high performance processors are designed and optimized by hand.*
  2. Low time-to-market and complex features $\Rightarrow$ It is preferable to avoid dealing with low-level details $\Rightarrow$ **high abstraction level**
     - *Example: SoC (System on Chip) architectures embed multiple heterogeneous components.*
- ForSyDe's main motivation is to solve the resulting abstraction gap problem

The abstraction-gap problem

- Systems designed nowadays (and electronic components in particular) are increasingly complex.
- Market pressure calls for two appartenly opposite constraints:
  1. Efficiency ⇒ It is required to handle low-level details at design time ⇒ **low abstraction level**
     - *Example: Large layout parts of today's high performance processors are designed and optimized by hand.*
  2. Low time-to-market and complex features ⇒ It is preferable to avoid dealing with low-level details ⇒ **high abstraction level**
     - *Example: SoC (System on Chip) architectures embed multiple heterogeneous components.*
- ForSyDe's main motivation is to solve the resulting abstraction gap problem

The abstraction-gap problem

- Systems designed nowadays (and electronic components in particular) are increasingly complex.
- Market pressure calls for two appartenly opposite constraints:
  1. Efficiency $\Rightarrow$ It is required to handle low-level details at design time $\Rightarrow$ **low abstraction level**
     - *Example: Large layout parts of today's high performance processors are designed and optimized by hand.*
  2. Low time-to-market and complex features $\Rightarrow$ It is preferable to avoid dealing with low-level details $\Rightarrow$ **high abstraction level**
     - *Example: SoC (System on Chip) architectures embed multiple heterogeneous components.*
- ForSyDe's main motivation is to solve the resulting abstraction gap problem

The abstraction-gap problem

- Systems designed nowadays (and electronic components in particular) are increasingly complex.
- Market pressure calls for two appartenly opposite constraints:
  1. Efficiency $\Rightarrow$ It is required to handle low-level details at design time $\Rightarrow$ **low abstraction level**
     - *Example: Large layout parts of today's high performance processors are designed and optimized by hand.*
  2. Low time-to-market and complex features $\Rightarrow$ It is preferable to avoid dealing with low-level details $\Rightarrow$ **high abstraction level**
     - *Example: SoC (System on Chip) architectures embed multiple heterogeneous components.*
- ForSyDe's main motivation is to solve the resulting abstraction gap problem

The abstraction-gap problem

- Systems designed nowadays (and electronic components in particular) are increasingly complex.
- Market pressure calls for two appartenly opposite constraints:
  1. Efficiency ⇒ It is required to handle low-level details at design time ⇒ **low abstraction level**
     - *Example: Large layout parts of today's high performance processors are designed and optimized by hand.*
  2. Low time-to-market and complex features ⇒ It is preferable to avoid dealing with low-level details ⇒ **high abstraction level**
     - *Example: SoC (System on Chip) architectures embed multiple heterogeneous components.*
- ForSyDe's main motivation is to solve the resulting abstraction gap problem

The abstraction-gap problem

- Systems designed nowadays (and electronic components in particular) are increasingly complex.
- Market pressure calls for two appartenly opposite constraints:
  1. Efficiency ⇒ It is required to handle low-level details at design time ⇒ **low abstraction level**
     - *Example: Large layout parts of today's high performance processors are designed and optimized by hand.*
  2. Low time-to-market and complex features ⇒ It is preferable to avoid dealing with low-level details ⇒ **high abstraction level**
     - *Example: SoC (System on Chip) architectures embed multiple heterogeneous components.*
- ForSyDe's main motivation is to solve the resulting abstraction gap problem

Outline

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a process constructor.
    - These parameters determine the behaviour of the process w.r.t. its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
  - .. are created from ***process constructors***.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a *process constructor*.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a *process constructor*.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a ***process constructor***.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a ***process constructor***.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a ***process constructor***.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a ***process constructor***.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a ***process constructor***.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a ***process constructor***.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a ***process constructor***.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a ***process constructor***.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - To simplify this talk, a Synchronous MoC will be assumed

What is ForSyDe?

- ForSyDe (Formal System Design) ..
  - .. is a System Design methodology
  - .. models Systems at a high abstraction level.
  - .. bridges the abstraction gap using a technique called ***transformational design refinement***.
- A system in ForSyDe is modelled as a network of cooperating ***processes*** which
  - .. are communicated via ***signals***.
    - Processes perform computations over its input signals and forward the results to adjacent processes through output signals.
  - .. are created from ***process constructors***.
    - The creation of a process entails setting the parameters (values or functions) of a ***process constructor***.
    - Those parameters determine the behaviour of the process over its input signals.
- Communication and computation are separated
  - Thus, time is abstracted, allowing multiple models of computation (Synchronous, Untimed, Discrete Time and Continuous).
  - **To simplify this talk, a Synchronous MoC will be assumed**

- **Signal**
  - A (possibly infinite) sequence of events, were each event has a tag and a value.
  - All events in a signal must have values of the same type.
  - In ForSyDe, signals are modelled as lists of event values. Tags are implicitly determined by the location of the values in the list.

$$\overrightarrow{s} = \ll v_0, v_1, v_2, \cdots \gg$$

  - The interpretation of tags depends on the model of computation. (e.g. in general, identical tags in different signals don't imply equal times)
  - In the Synchronous MoC
    - the system is governed by a global clock
    - tags correspond to global-clock cycles (i.e each value corresponds to a clock cycle).

- **Signal**
  - A (possibly infinite) sequence of events, were each event has a tag and a value.
  - All events in a signal must have values of the same type.
  - In ForSyDe, signals are modelled as lists of event values. Tags are implicitly determined by the location of the values in the list.

    $$\vec{s} = \ll v_0, v_1, v_2, \cdots \gg$$

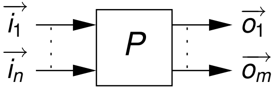
  - The interpretation of tags depends on the model of computation. (e.g. in general, identical tags in different signals don't imply equal times)
  - In the Synchronous MoC
    - the system is governed by a global clock
    - tags correspond to global-clock cycles (i.e each value corresponds to a clock cycle).

- **Signal**
  - A (possibly infinite) sequence of events, were each event has a tag and a value.
  - All events in a signal must have values of the same type.
  - In ForSyDe, signals are modelled as lists of event values. Tags are implicitly determined by the location of the values in the list.

    $$\vec{s} = \ll v_0, v_1, v_2, \cdots \gg$$

  - The interpretation of tags depends on the model of computation. (e.g. in general, identical tags in different signals don't imply equal times)
  - In the Synchronous MoC
    - the system is governed by a global clock
    - tags correspond to global-clock cycles (i.e each value corresponds to a clock cycle).

- **Process**



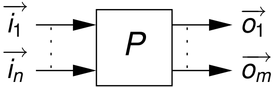
  - Processes are defined as **pure** functions over signals.
    - $p : \underbrace{S \times S \times \ldots \times S}_{n} \to \underbrace{S \times S \times \ldots \times S}_{m}$
    - $i_0 = i'_0, i_1 = i'_1, \ldots i_n = i'_n \Rightarrow p(i_1, i_2, \ldots, i_n) = p(i'_1, i'_2, \ldots, i'_n)$

- **Process constructor**
  - Creates a processes out of:
    - Values: process configuration parameter or initial state.
    - Functions: process behaviour.
    - Note: These functions operate over the values carried by signals, not over signals themselves.
    - $p = pc(v_1, v_2, \ldots, f_1, f_2, \ldots)$

Key concepts (II)

Processes and Process Constructors

- **Process**



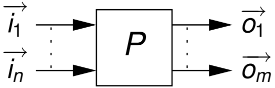
  - Processes are defined as **pure** functions over signals.
    - $p : \underbrace{S \times S \times \ldots \times S}_{n} \to \underbrace{S \times S \times \ldots \times S}_{m}$
    - $i_0 = i'_0, i_1 = i'_1, \ldots i_n = i'_n \Rightarrow p(i_1, i_2, \ldots, i_n) = p(i'_1, i'_2, \ldots, i'_n)$

- **Process constructor**
  - Creates a processes out of:
    - Values: process configuration parameter or initial state.
    - Functions: process behaviour.
    - Note: These functions operate over the values carried by signals, not over signals themselves.
    - $p = pc(v_1, v_2, \ldots, f_1, f_2, \ldots)$

Key concepts (II)

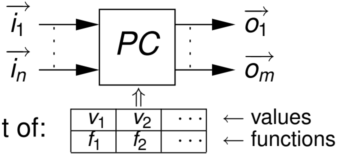
Processes and Process Constructors

- **Process**



  - Processes are defined as **pure** functions over signals.
    - $p : \underbrace{S \times S \times \ \dots \ \times \ S}_{n} \rightarrow \underbrace{S \times S \times \ \dots \ \times \ S}_{m}$
    - $i_0 = i'_0, i_1 = i'_1, \dots i_n = i'_n \Rightarrow p(i_1, i_2, \dots, i_n) = p(i'_1, i'_2, \dots, i'_n)$

- **Process constructor**



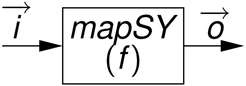
  - Creates a processes out of:
    - Values: process configuration parameter or initial state.
    - Functions: process behaviour.
    - Note: These functions operate over the values carried by signals, not over signals themselves.
    - $p = pc(v_1, v_2, \dots, f_1, f_2, \dots)$

Outline

1. Introduction to ForSyDe
   - The abstraction gap problem
   - What is ForSyDe?
   - Examples
   - Simplified Design Flow
   - Initial implementation of ForSyDe
2. Current ForSyDe's implementation details
   - The compilation problem
   - Deep Embedding + Embedded compiler
   - The Sharing Problem: Observable Sharing
   - The Polymorphism Problem: Dynamic Types
   - Deep-embedding process-constructor parameters
   - Components

Primitive Process-Constructor Examples

- *mapSY* primitive process-constructor



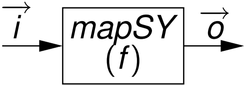
$$mapSY(f)(\ll v_0, v_1, v_2, \cdots \gg) = \ll f(v_0), f(v_1), f(v_2), \cdots \gg$$

- *delaySY$_k$* primitive process constructor

$$delaySY_k(\ll v_0, v_1, v_2, \cdots \gg) = \ll \underbrace{s_0, s_0, s_0, \ldots}_{k}, v_0, v_1, v_2, \cdots \gg$$

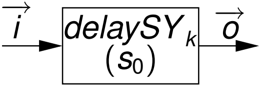
Primitive Process-Constructor Examples

- *mapSY* primitive process-constructor



$$mapSY(f)(\ll v_0, v_1, v_2, \cdots \gg) = \ll f(v_0), f(v_1), f(v_2), \cdots \gg$$

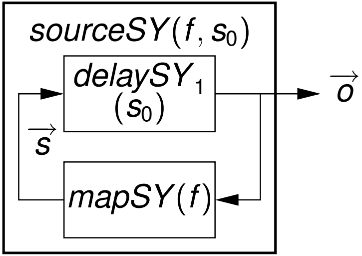
- $delaySY_k$ primitive process constructor



$$delaySY_k(\ll v_0, v_1, v_2, \cdots \gg) = \ll \underbrace{s_0, s_0, s_0, \ldots,}_{k} v_0, v_1, v_2, \cdots \gg$$

More Examples

- The *sourceSY* derived process constructor



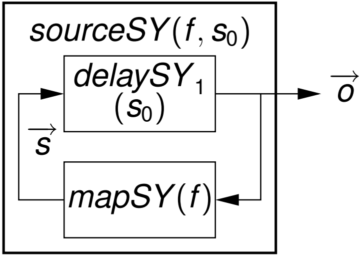
$$sourceSY(f, s_0) = \ll s_0, f(s_0), f(f(s_0)), f(f(f(s_0))), \cdots \gg$$

- Sample use of a process constructor: the trivial *plus*1 process

$$plus1 = mapSY(+1)$$

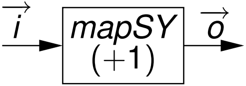
More Examples

- The *sourceSY* derived process constructor



$$sourceSY(f, s_0) = \ll s_0, f(s_0), f(f(s_0)), f(f(f(s_0))), \cdots \gg$$

- Sample use of a process constructor: the trivial *plus*1 process

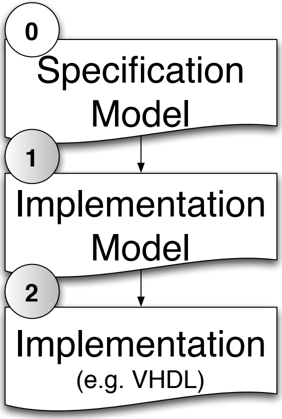


$$plus1 = mapSY(+1)$$

Outline

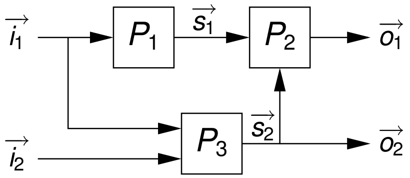
1. Introduction to ForSyDe
   - The abstraction gap problem
   - What is ForSyDe?
   - Examples
   - Simplified Design Flow
   - Initial implementation of ForSyDe
2. Current ForSyDe's implementation details
   - The compilation problem
   - Deep Embedding + Embedded compiler
   - The Sharing Problem: Observable Sharing
   - The Polymorphism Problem: Dynamic Types
   - Deep-embedding process-constructor parameters
   - Components

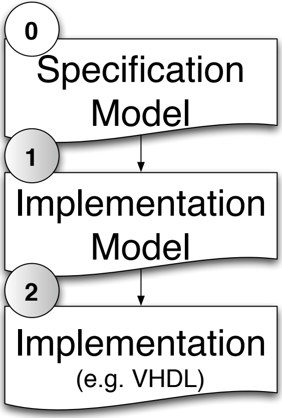
Simplified Design Flow



Simplified Design Flow

0) The designer creates the **specification model** as a network of processes.

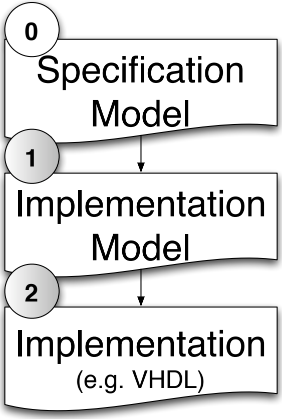




Simplified Design Flow

1) **Transformational refinement**

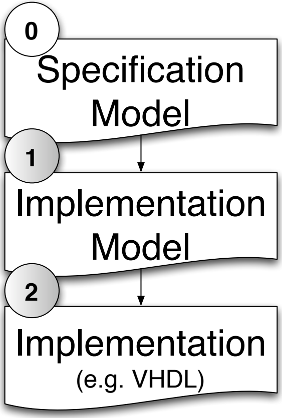
- The **specification model** is transformed into a lower-level **implementation model**
- This stage is in charge of bridging the abstraction gap using transformation rules.
- Rules can be
  - semantic preserving (automatic)
  - non-semantic preserving (require interaction with the designer).
- Relies on the formal foundations of ForSyDe.
- Theoretically designed but not yet implemented.



Simplified Design Flow

2) **Implementation Mapping**

- Transforms the **implementation model** into an architecture-specific implementation
  - Software: C, C++ . . .
  - Hardware: VHDL, SystemC, Verilog . . .
  - Special cases: Simulation, Verification.
- Current lack of automatization of (1) entails working with the **specification model** directly
- Implemented mappings: Simulation, VHDL (in progress)



Outline

1. Introduction to ForSyDe
   - The abstraction gap problem
   - What is ForSyDe?
   - Examples
   - Simplified Design Flow
   - Initial implementation of ForSyDe
2. Current ForSyDe's implementation details
   - The compilation problem
   - Deep Embedding + Embedded compiler
   - The Sharing Problem: Observable Sharing
   - The Polymorphism Problem: Dynamic Types
   - Deep-embedding process-constructor parameters
   - Components

Initial implementation of ForSyDe (I)

- The specification models were initially expressed in a **shallow** Haskell-embedded DSL (*Domain Specific Language*)
- A DSL, as opposed to a general purpose programming language (C,C++,Ada, Haskell) is designed for a specific task.
  - Examples: YACC, Postscript, GraphViz, VHDL ...
- An Embedded DSL is implemented inside a general purpose programming language (called host language), as a library.
- The embedding can be shallow or deep
  - Shallow: The data structures supporting the embedded language only reflect semantics.
  - Deep: The data structures supporting the embedded language reflect the structure of the program which created them.
- The embedded approach has some advantages and disadvantages
  - The host language plus all its surrounding machinery (compilers, libraries ..) can be reused.
  - The syntax and semantics of the two languages (embedded and host) might differ.

Initial implementation of ForSyDe (I)

- The specification models were initially expressed in a **shallow** Haskell-embedded DSL (*Domain Specific Language*)
- A DSL, as opposed to a general purpose programming language (C,C++,Ada, Haskell) is designed for a specific task.
  - Examples: YACC, Postscript, GraphViz, VHDL ...
- An Embedded DSL is implemented inside a general purpose programming language (called host language), as a library.
- The embedding can be shallow or deep
  - Shallow: The data structures supporting the embedded language only reflect semantics.
  - Deep: The data structures supporting the embedded language reflect the structure of the program which created them.
- The embedded approach has some advantages and disadvantages
  - The host language plus all its surrounding machinery (compilers, libraries ..) can be reused.
  - The syntax and semantics of the two languages (embedded and host) might differ.

Initial implementation of ForSyDe (I)

- The specification models were initially expressed in a **shallow** Haskell-embedded DSL (*Domain Specific Language*)
- A DSL, as opposed to a general purpose programming language (C,C++,Ada, Haskell) is designed for a specific task.
  - Examples: YACC, Postscript, GraphViz, VHDL ...
- An Embedded DSL is implemented inside a general purpose programming language (called host language), as a library.
- The embedding can be shallow or deep
  - Shallow: The data structures supporting the embedded language only reflect semantics.
  - Deep: The data structures supporting the embedded language reflect the structure of the program which created them.
- The embedded approach has some advantages and disadvantages
  - The host language plus all its surrounding machinery (compilers, libraries ..) can be reused.
  - The syntax and semantics of the two languages (embedded and host) might differ.

Initial implementation of ForSyDe (I)

- The specification models were initially expressed in a **shallow** Haskell-embedded DSL (*Domain Specific Language*)
- A DSL, as opposed to a general purpose programming language (C,C++,Ada, Haskell) is designed for a specific task.
  - Examples: YACC, Postscript, GraphViz, VHDL ...
- An Embedded DSL is implemented inside a general purpose programming language (called host language), as a library.
- The embedding can be shallow or deep
  - Shallow: The data structures supporting the embedded language only reflect semantics.
  - Deep: The data structures supporting the embedded language reflect the structure of the program which created them.
- The embedded approach has some advantages and disadvantages
  - The host language plus all its surrounding machinery (compilers, libraries ..) can be reused.
  - The syntax and semantics of the two languages (embedded and host) might differ.

Initial implementation of ForSyDe (I)

- The specification models were initially expressed in a **shallow** Haskell-embedded DSL (*Domain Specific Language*)
- A DSL, as opposed to a general purpose programming language (C,C++,Ada, Haskell) is designed for a specific task.
  - Examples: YACC, Postscript, GraphViz, VHDL ...
- An Embedded DSL is implemented inside a general purpose programming language (called host language), as a library.
- The embedding can be shallow or deep
  - Shallow: The data structures supporting the embedded language only reflect semantics.
  - Deep: The data structures supporting the embedded language reflect the structure of the program which created them.
- The embedded approach has some advantages and disadvantages
  - The host language plus all its surrounding machinery (compilers, libraries ..) can be reused.
  - The syntax and semantics of the two languages (embedded and host) might differ.

Initial implementation of ForSyDe (I)

- The specification models were initially expressed in a **shallow** Haskell-embedded DSL (*Domain Specific Language*)
- A DSL, as opposed to a general purpose programming language (C,C++,Ada, Haskell) is designed for a specific task.
  - Examples: YACC, Postscript, GraphViz, VHDL ...
- An Embedded DSL is implemented inside a general purpose programming language (called host language), as a library.
- The embedding can be shallow or deep
  - Shallow: The data structures supporting the embedded language only reflect semantics.
  - Deep: The data structures supporting the embedded language reflect the structure of the program which created them.
- The embedded approach has some advantages and disadvantages
  - The host language plus all its surrounding machinery (compilers, libraries ..) can be reused.
  - The syntax and semantics of the two languages (embedded and host) might differ.

Initial implementation of ForSyDe (I)

- The specification models were initially expressed in a **shallow** Haskell-embedded DSL (*Domain Specific Language*)
- A DSL, as opposed to a general purpose programming language (C,C++,Ada, Haskell) is designed for a specific task.
  - Examples: YACC, Postscript, GraphViz, VHDL ...
- An Embedded DSL is implemented inside a general purpose programming language (called host language), as a library.
- The embedding can be shallow or deep
  - Shallow: The data structures supporting the embedded language only reflect semantics.
  - Deep: The data structures supporting the embedded language reflect the structure of the program which created them.
- The embedded approach has some advantages and disadvantages
  - The host language plus all its surrounding machinery (compilers, libraries ..) can be reused.
  - The syntax and semantics of the two languages (embedded and host) might differ.

Initial implementation of ForSyDe (I)

- The specification models were initially expressed in a **shallow** Haskell-embedded DSL (*Domain Specific Language*)
- A DSL, as opposed to a general purpose programming language (C,C++,Ada, Haskell) is designed for a specific task.
  - Examples: YACC, Postscript, GraphViz, VHDL ...
- An Embedded DSL is implemented inside a general purpose programming language (called host language), as a library.
- The embedding can be shallow or deep
  - Shallow: The data structures supporting the embedded language only reflect semantics.
  - Deep: The data structures supporting the embedded language reflect the structure of the program which created them.
- The embedded approach has some advantages and disadvantages
  - The host language plus all its surrounding machinery (compilers, libraries ..) can be reused.
  - The syntax and semantics of the two languages (embedded and host) might differ.

Initial implementation of ForSyDe (I)

- The specification models were initially expressed in a **shallow** Haskell-embedded DSL (*Domain Specific Language*)
- A DSL, as opposed to a general purpose programming language (C,C++,Ada, Haskell) is designed for a specific task.
  - Examples: YACC, Postscript, GraphViz, VHDL ...
- An Embedded DSL is implemented inside a general purpose programming language (called host language), as a library.
- The embedding can be shallow or deep
  - Shallow: The data structures supporting the embedded language only reflect semantics.
  - Deep: The data structures supporting the embedded language reflect the structure of the program which created them.
- The embedded approach has some advantages and disadvantages
  - The host language plus all its surrounding machinery (compilers, libraries ..) can be reused.
  - The syntax and semantics of the two languages (embedded and host) might differ.

Initial implementation of ForSyDe (I)

- The specification models were initially expressed in a **shallow** Haskell-embedded DSL (*Domain Specific Language*)
- A DSL, as opposed to a general purpose programming language (C,C++,Ada, Haskell) is designed for a specific task.
  - Examples: YACC, Postscript, GraphViz, VHDL ...
- An Embedded DSL is implemented inside a general purpose programming language (called host language), as a library.
- The embedding can be shallow or deep
  - Shallow: The data structures supporting the embedded language only reflect semantics.
  - Deep: The data structures supporting the embedded language reflect the structure of the program which created them.
- The embedded approach has some advantages and disadvantages
  - The host language plus all its surrounding machinery (compilers, libraries ..) can be reused.
  - The syntax and semantics of the two languages (embedded and host) might differ.

Initial implementation of ForSyDe (II)

- Why Haskell?
  - Strongly-typed: DSLs are easy to embed.
  - Lazy: infinite data structures are natively supported.
    - Signals can be easily shallow-embedded with a type isomorphic to lists.

```haskell
data Signal a = NullS | a :- Signal a
```

  - Purely functional with higher-order functions:
    - Process constructors are just pure, higher-order functions after all.

```haskell
mapSY :: (a -> b) -> Signal a -> Signal b
mapSY _ NullS = NullS
mapSY f (x:-xs) = f x :- (mapSY f xs)
```

    - Haskell is particularly good for creating function combinators, useful to create process connection patterns.

Initial implementation of ForSyDe (II)

- Why Haskell?
  - Strongly-typed: DSLs are easy to embed.
  - Lazy: infinite data structures are natively supported.
    - Signals can be easily shallow-embedded with a type isomorphic to lists.

```
data Signal a = NullS | a :- Signal a
```

  - Purely functional with higher-order functions:
    - Process constructors are just pure, higher-order functions after all.

```
mapSY :: (a -> b) -> Signal a -> Signal b
mapSY _ NullS = NullS
mapSY f (x:-xs) = f x :- (mapSY f xs)
```

    - Haskell is particularly good for creating function combinators, useful to create process connection patterns.

Initial implementation of ForSyDe (II)

- Why Haskell?
  - Strongly-typed: DSLs are easy to embed.
  - Lazy: infinite data structures are natively supported.
    - Signals can be easily shallow-embedded with a type isomorphic to lists.

```haskell
data Signal a = NullS | a :- Signal a
```

  - Purely functional with higher-order functions:
    - Process constructors are just pure, higher-order functions after all.

```haskell
mapSY :: (a -> b) -> Signal a -> Signal b
mapSY _ NullS = NullS
mapSY f (x:-xs) = f x :- (mapSY f xs)
```

    - Haskell is particularly good for creating function combinators, useful to create process connection patterns.

Initial implementation of ForSyDe (II)

- Why Haskell?
  - Strongly-typed: DSLs are easy to embed.
  - Lazy: infinite data structures are natively supported.
    - Signals can be easily shallow-embedded with a type isomorphic to lists.

```haskell
data Signal a = NullS | a :- Signal a
```

  - Purely functional with higher-order functions:
    - Process constructors are just pure, higher-order functions after all.

```haskell
mapSY :: (a -> b) -> Signal a -> Signal b
mapSY _ NullS = NullS
mapSY f (x:-xs) = f x :- (mapSY f xs)
```

    - Haskell is particularly good for creating function combinators, useful to create process connection patterns.

Outline

Outline

The compilation problem

- The initial **shallow** embedding only allows simulating the models (The `Signal` type is not aware of how the system was built).
- We want to be able to implement the transformational refinement and implementation mapping stages without losing the ability of simulating our models.
- Alternatives
  - Standalone Haskell compiler: excessive and unfeasible given our development resources.
  - Creating a backend for an existing compiler: slightly less excessive and unfeasible.
  - Keep the shallow embedding for simulation and use a static analyzer for compilation: very difficult without restricting how the host language is used.
  - Deep embedding plus embedded compiler: chosen solution.

The compilation problem

- The initial **shallow** embedding only allows simulating the models (The `Signal` type is not aware of how the system was built).
- We want to be able to implement the transformational refinement and implementation mapping stages without losing the ability of simulating our models.
- Alternatives
  - Standalone Haskell compiler: excessive and unfeasible given our development resources.
  - Creating a backend for an existing compiler: slightly less excessive and unfeasible.
  - Keep the shallow embedding for simulation and use a static analyzer for compilation: very difficult without restricting how the host language is used.
  - Deep embedding plus embedded compiler: chosen solution.

The compilation problem

- The initial **shallow** embedding only allows simulating the models (The `Signal` type is not aware of how the system was built).
- We want to be able to implement the transformational refinement and implementation mapping stages without losing the ability of simulating our models.
- Alternatives
  - Standalone Haskell compiler: excessive and unfeasible given our development resources.
  - Creating a backend for an existing compiler: slightly less excessive and unfeasible.
  - Keep the shallow embedding for simulation and use a static analyzer for compilation: very difficult without restricting how the host language is used.
  - Deep embedding plus embedded compiler: chosen solution.

The compilation problem

- The initial **shallow** embedding only allows simulating the models (The `Signal` type is not aware of how the system was built).
- We want to be able to implement the transformational refinement and implementation mapping stages without losing the ability of simulating our models.
- Alternatives
  - Standalone Haskell compiler: excessive and unfeasible given our development resources.
  - Creating a backend for an existing compiler: slightly less excessive and unfeasible.
  - Keep the shallow embedding for simulation and use a static analyzer for compilation: very difficult without restricting how the host language is used.
  - Deep embedding plus embedded compiler: chosen solution.

The compilation problem

- The initial **shallow** embedding only allows simulating the models (The `Signal` type is not aware of how the system was built).
- We want to be able to implement the transformational refinement and implementation mapping stages without losing the ability of simulating our models.
- Alternatives
  - Standalone Haskell compiler: excessive and unfeasible given our development resources.
  - Creating a backend for an existing compiler: slightly less excessive and unfeasible.
  - Keep the shallow embedding for simulation and use a static analyzer for compilation: very difficult without restricting how the host language is used.
  - Deep embedding plus embedded compiler: chosen solution.

The compilation problem

- The initial **shallow** embedding only allows simulating the models (The `Signal` type is not aware of how the system was built).
- We want to be able to implement the transformational refinement and implementation mapping stages without losing the ability of simulating our models.
- Alternatives
  - Standalone Haskell compiler: excessive and unfeasible given our development resources.
  - Creating a backend for an existing compiler: slightly less excessive and unfeasible.
  - Keep the shallow embedding for simulation and use a static analyzer for compilation: very difficult without restricting how the host language is used.
  - Deep embedding plus embedded compiler: chosen solution.

The compilation problem

- The initial **shallow** embedding only allows simulating the models (The `Signal` type is not aware of how the system was built).
- We want to be able to implement the transformational refinement and implementation mapping stages without losing the ability of simulating our models.
- Alternatives
  - Standalone Haskell compiler: excessive and unfeasible given our development resources.
  - Creating a backend for an existing compiler: slightly less excessive and unfeasible.
  - Keep the shallow embedding for simulation and use a static analyzer for compilation: very difficult without restricting how the host language is used.
  - Deep embedding plus embedded compiler: chosen solution.

Outline

Deep Embedding + Embedded compiler

- Goal: Create a new deep-embedded `Signal`, aware of of the system structure.
  - The new `Signal` will be the intermediate representation of an embedded compiler, in charge of the implementation mapping.
- A possible simplified solution (for a structural hardware design language).

```
data Signal = Comp String [Signal]
inv, latch :: Signal -> Signal -- sample primitives
inv s = Comp "inv" [b] -- inverter primitive
latch s = Comp "latch" [s] -- latch primitive
toggle :: Signal -- sample circuit
toggle = let o = inv (latch o) in o
```

- Problems:
  - Detecting sharing between components (there is no way to detect the loop in `toggle`).
  - ForSyDe signals are Polymorphic: how to represent polymorphism?
  - ForSyDe is behavioural: how to store functions in `Signal` for later translation?

Deep Embedding + Embedded compiler

- Goal: Create a new deep-embedded `Signal`, aware of of the system structure.
  - The new `Signal` will be the intermediate representation of an embedded compiler, in charge of the implementation mapping.
- A possible simplified solution (for a structural hardware design language).

```
data Signal = Comp String [Signal]
inv, latch :: Signal -> Signal -- sample primitives
inv s = Comp "inv" [b] -- inverter primitive
latch s = Comp "latch" [s] -- latch primitive
toggle :: Signal -- sample circuit
toggle = let o = inv (latch o) in o
```

- Problems:
  - Detecting sharing between components (there is no way to detect the loop in `toggle`).
  - ForSyDe signals are Polymorphic: how to represent polymorphism?
  - ForSyDe is behavioural: how to store functions in `Signal` for later translation?

Deep Embedding + Embedded compiler

- Goal: Create a new deep-embedded `Signal`, aware of of the system structure.
  - The new `Signal` will be the intermediate representation of an embedded compiler, in charge of the implementation mapping.
- A possible simplified solution (for a structural hardware design language).

```
data Signal = Comp String [Signal]
inv, latch :: Signal -> Signal -- sample primitives
inv s = Comp "inv" [b] -- inverter primitive
latch s = Comp "latch" [s] -- latch primitive
toggle :: Signal -- sample circuit
toggle = let o = inv (latch o) in o
```

- Problems:
  - Detecting sharing between components (there is no way to detect the loop in `toggle`).
  - ForSyDe signals are Polymorphic: how to represent polymorphism?
  - ForSyDe is behavioural: how to store functions in `Signal` for later translation?

Deep Embedding + Embedded compiler

- Goal: Create a new deep-embedded `Signal`, aware of of the system structure.
  - The new `Signal` will be the intermediate representation of an embedded compiler, in charge of the implementation mapping.
- A possible simplified solution (for a structural hardware design language).

```
data Signal = Comp String [Signal]
inv, latch :: Signal -> Signal -- sample primitives
inv s = Comp "inv" [b] -- inverter primitive
latch s = Comp "latch" [s] -- latch primitive
toggle :: Signal -- sample circuit
toggle = let o = inv (latch o) in o
```

- Problems:
  - Detecting sharing between components (there is no way to detect the loop in `toggle`).
  - ForSyDe signals are Polymorphic: how to represent polymorphism?
  - ForSyDe is behavioural: how to store functions in `Signal` for later translation?

Deep Embedding + Embedded compiler

- Goal: Create a new deep-embedded `Signal`, aware of of the system structure.
  - The new `Signal` will be the intermediate representation of an embedded compiler, in charge of the implementation mapping.
- A possible simplified solution (for a structural hardware design language).

```
data Signal = Comp String [Signal]
inv, latch :: Signal -> Signal -- sample primitives
inv s = Comp "inv" [b] -- inverter primitive
latch s = Comp "latch" [s] -- latch primitive
toggle :: Signal -- sample circuit
toggle = let o = inv (latch o) in o
```

- Problems:
  - Detecting sharing between components (there is no way to detect the loop in `toggle`).
  - ForSyDe signals are Polymorphic: how to represent polymorphism?
  - ForSyDe is behavioural: how to store functions in `Signal` for later translation?

Deep Embedding + Embedded compiler

- Goal: Create a new deep-embedded `Signal`, aware of of the system structure.
  - The new `Signal` will be the intermediate representation of an embedded compiler, in charge of the implementation mapping.
- A possible simplified solution (for a structural hardware design language).

```haskell
data Signal = Comp String [Signal]
inv, latch :: Signal -> Signal -- sample primitives
inv s = Comp "inv" [b] -- inverter primitive
latch s = Comp "latch" [s] -- latch primitive
toggle :: Signal -- sample circuit
toggle = let o = inv (latch o) in o
```

- Problems:
  - Detecting sharing between components (there is no way to detect the loop in `toggle`).
  - ForSyDe signals are Polymorphic: how to represent polymorphism?
  - ForSyDe is behavioural: how to store functions in `Signal` for later translation?

Outline

The Sharing Problem: Observable Sharing

- Explicit tagging: the designer provides a unique label for each component.

```
type Label = String
data Signal = Comp Label String [Signal]
toggle = let o = inv "tinv" (latch "tlatch" o) in o
```

  - Artificial tag syntax, uniqueness is not guaranteed.
- Transform `Signal` into a monad which generates unique labels.
  - Guaranteed uniqueness but inconvenient monadic syntax.
- Observable sharing: link components through unmutable references

```
data Ref a = Ref (IORef a) deriving Eq
newRef = Ref.unsafePerformIO.newIORef
data Signal = Comp String [Ref Signal]
```

  - Pros: Guaranteed uniqueness without inconvenient syntax
  - Cons: Impure extension
    - However, referential transparency will be preserved if sharing is (all known Haskell compilers are based in graph reduction).

The Sharing Problem: Observable Sharing

- Explicit tagging: the designer provides a unique label for each component.

```haskell
type Label = String
data Signal = Comp Label String [Signal]
toggle = let o = inv "tinv" (latch "tlatch" o) in o
```

  - Artificial tag syntax, uniqueness is not guaranteed.
- Transform `Signal` into a monad which generates unique labels.
  - Guaranteed uniqueness but inconvenient monadic syntax.
- Observable sharing: link components through unmutable references

```haskell
data Ref a = Ref (IORef a) deriving Eq
newRef = Ref.unsafePerformIO.newIORef
data Signal = Comp String [Ref Signal]
```

  - Pros: Guaranteed uniqueness without inconvenient syntax
  - Cons: Impure extension
    - However, referential transparency will be preserved if sharing is (all known Haskell compilers are based in graph reduction).

The Sharing Problem: Observable Sharing

- Explicit tagging: the designer provides a unique label for each component.

```
type Label = String
data Signal = Comp Label String [Signal]
toggle = let o = inv "tinv" (latch "tlatch" o) in o
```

  - Artificial tag syntax, uniqueness is not guaranteed.
- Transform `Signal` into a monad which generates unique labels.
  - Guaranteed uniqueness but inconvenient monadic syntax.
- Observable sharing: link components through unmutable references

```
data Ref a = Ref (IORef a) deriving Eq
newRef = Ref.unsafePerformIO.newIORef
data Signal = Comp String [Ref Signal]
```

  - Pros: Guaranteed uniqueness without inconvenient syntax
  - Cons: Impure extension
    - However, referential transparency will be preserved if sharing is (all known Haskell compilers are based in graph reduction).

The Sharing Problem: Observable Sharing

- Explicit tagging: the designer provides a unique label for each component.

```
type Label = String
data Signal = Comp Label String [Signal]
toggle = let o = inv "tinv" (latch "tlatch" o) in o
```

  - Artificial tag syntax, uniqueness is not guaranteed.
- Transform `Signal` into a monad which generates unique labels.
  - Guaranteed uniqueness but inconvenient monadic syntax.
- Observable sharing: link components through unmutable references

```
data Ref a = Ref (IORef a) deriving Eq
newRef = Ref.unsafePerformIO.newIORef
data Signal = Comp String [Ref Signal]
```

  - Pros: Guaranteed uniqueness without inconvenient syntax
  - Cons: Impure extension
    - However, referential transparency will be preserved if sharing is (all known Haskell compilers are based in graph reduction).

The Sharing Problem: Observable Sharing

- Explicit tagging: the designer provides a unique label for each component.

```
type Label = String
data Signal = Comp Label String [Signal]
toggle = let o = inv "tinv" (latch "tlatch" o) in o
```

  - Artificial tag syntax, uniqueness is not guaranteed.
- Transform `Signal` into a monad which generates unique labels.
  - Guaranteed uniqueness but inconvenient monadic syntax.
- Observable sharing: link components through unmutable references

```
data Ref a = Ref (IORef a) deriving Eq
newRef = Ref.unsafePerformIO.newIORef
data Signal = Comp String [Ref Signal]
```

  - Pros: Guaranteed uniqueness without inconvenient syntax
  - Cons: Impure extension
    - However, referential transparency will be preserved if sharing is (all known Haskell compilers are based in graph reduction).

The Sharing Problem: Observable Sharing

- Explicit tagging: the designer provides a unique label for each component.

```
type Label = String
data Signal = Comp Label String [Signal]
toggle = let o = inv "tinv" (latch "tlatch" o) in o
```

  - Artificial tag syntax, uniqueness is not guaranteed.
- Transform `Signal` into a monad which generates unique labels.
  - Guaranteed uniqueness but inconvenient monadic syntax.
- Observable sharing: link components through unmutable references

```
data Ref a = Ref (IORef a) deriving Eq
newRef = Ref.unsafePerformIO.newIORef
data Signal = Comp String [Ref Signal]
```

  - Pros: Guaranteed uniqueness without inconvenient syntax
  - Cons: Impure extension
    - However, referential transparency will be preserved if sharing is (all known Haskell compilers are based in graph reduction).

Outline

The Polymorphism Problem: Dynamic Types

- So far we have solved the sharing problem, but how to take polymorphism in account?

```
-- phantom parameter to ensure type-consistency
data Signal a = Signal PrimSignal
-- first attempt, incorrect
data PrimSignal = MapSY (a->b) (Ref PrimSignal) ...
mapSY :: (a->b) -> Signal a -> Signal b
```

- Solution: Dynamic types.

```
class Typeable a where
  typeOf :: a -> TypeRep
toDyn :: Typeable a => a -> Dynamic
fromDynamic :: Typeable a => Dynamic -> Maybe a

-- correct
data PrimSignal = MapSY Dynamic (Ref PrimSignal) ...
mapSY :: (Typeable a, Typeable b) =>
         (a->b) -> Signal a -> Signal b
```

The Polymorphism Problem: Dynamic Types

- So far we have solved the sharing problem, but how to take polymorphism in account?

```haskell
-- phantom parameter to ensure type-consistency
data Signal a = Signal PrimSignal
-- first attempt, incorrect
data PrimSignal = MapSY (a->b) (Ref PrimSignal) ...
mapSY :: (a->b) -> Signal a -> Signal b
```

- Solution: Dynamic types.

```haskell
class Typeable a where
  typeOf :: a -> TypeRep
toDyn :: Typeable a => a -> Dynamic
fromDynamic :: Typeable a => Dynamic -> Maybe a

-- correct
data PrimSignal = MapSY Dynamic (Ref PrimSignal) ...
mapSY :: (Typeable a, Typeable b) =>
         (a->b) -> Signal a -> Signal b
```

Outline

Deep-embedding process-constructor parameters

- The value of the process constructor parameters is not enough, the compiler needs their AST for later translation.
- Solution: Template Haskell (compile-time metaprogramming extension)

```
[d| decs |] :: Q [Dec] -- lift the AST of the enclosed declarations
$(exp) -- splice declarations or expressions
-- exp must be a Haskell expression of type (Q Exp) or (Q [Dec])
```

```
-- the Dynamic value is kept for simulation
data PrimSignal = MapSY Dynamic [Dec] (Ref PrimSignal) ...
mapSY :: (Typeable a, Typeable b) =>
         ProcFun (a->b) -> Signal a -> Signal b
newProcFun :: Q [Dec] -> Q Exp
```

- Example: *plus*1

```
plus1 :: (Typeable a, Num a) => Signal a -> Signal a
plus1 = mapSY (+1)
```

Deep-embedding process-constructor parameters

- The value of the process constructor parameters is not enough, the compiler needs their AST for later translation.
- Solution: Template Haskell (compile-time metaprogramming extension)

```
[d| decs |] :: Q [Dec] -- lift the AST of the enclosed declarations
$(exp) -- splice declarations or expressions
-- exp must be a Haskell expression of type (Q Exp) or (Q [Dec])
```

```
-- the Dynamic value is kept for simulation
data PrimSignal = MapSY Dynamic [Dec] (Ref PrimSignal) ...
mapSY :: (Typeable a, Typeable b) =>
         ProcFun (a->b) -> Signal a -> Signal b
newProcFun :: Q [Dec] -> Q Exp
```

- Example: *plus*1

```
plus1 :: (Typeable a, Num a) => Signal a -> Signal a
plus1 = mapSY (+1)
```

Deep-embedding process-constructor parameters

- The value of the process constructor parameters is not enough, the compiler needs their AST for later translation.
- Solution: Template Haskell (compile-time metaprogramming extension)

```
[d| decs |] :: Q [Dec] -- lift the AST of the enclosed declarations
$(exp) -- splice declarations or expressions
-- exp must be a Haskell expression of type (Q Exp) or (Q [Dec])
```

```
-- the Dynamic value is kept for simulation
data PrimSignal = MapSY Dynamic [Dec] (Ref PrimSignal) ...
mapSY :: (Typeable a, Typeable b) =>
         ProcFun (a->b) -> Signal a -> Signal b
newProcFun :: Q [Dec] -> Q Exp
```

- Example: *plus*1

```
plus1 :: (Typeable a, Num a) => Signal a -> Signal a
plus1 = mapSY (+1)
```

Deep-embedding process-constructor parameters

- The value of the process constructor parameters is not enough, the compiler needs their AST for later translation.
- Solution: Template Haskell (compile-time metaprogramming extension)

```
[d| decs |] :: Q [Dec] -- lift the AST of the enclosed declarations
$(exp) -- splice declarations or expressions
-- exp must be a Haskell expression of type (Q Exp) or (Q [Dec])
```

```
-- the Dynamic value is kept for simulation
data PrimSignal = MapSY Dynamic [Dec] (Ref PrimSignal) ...
mapSY :: (Typeable a, Typeable b) =>
         ProcFun (a->b) -> Signal a -> Signal b
newProcFun :: Q [Dec] -> Q Exp
```

- Example: *plus*1

```
plus1 :: (Typeable a, Num a) => Signal a -> Signal a
plus1 = mapSY p1
 where p1 = $(newProcFun [d| doPlus1 :: Num a => a -> a
                              doPlus1 a = a + 1          |])
```

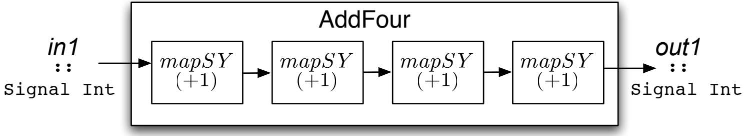
Outline

Components

- Similarly to HDLs like VHDL, ForSyDe has support for hierarchical design through components.
- Let's see a simple example. Design a serial adder using components in 5 simple steps.

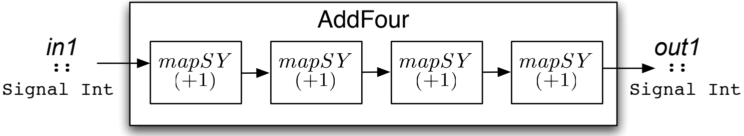
Components

- Similarly to HDLs like VHDL, ForSyDe has support for hierarchical design through components.
- Let's see a simple example. Design a serial adder using components in 5 simple steps.



Components

- Similarly to HDLs like VHDL, ForSyDe has support for hierarchical design through components.
- Let's see a simple example. Design a serial adder using components in 5 simple steps.

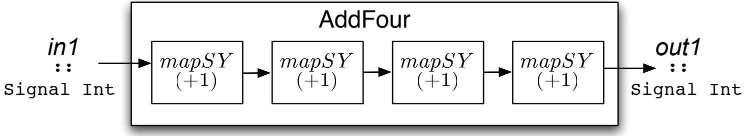


1) Create a process function which adds one to its input

```
addOnef :: ProcFun (Int32 -> Int32)
addOnef = $(newProcFun [d| addOnef :: Int32 -> Int32
                           addOnef n = n + 1 |])
```

Components

- Similarly to HDLs like VHDL, ForSyDe has support for hierarchical design through components.
- Let's see a simple example. Design a serial adder using components in 5 simple steps.

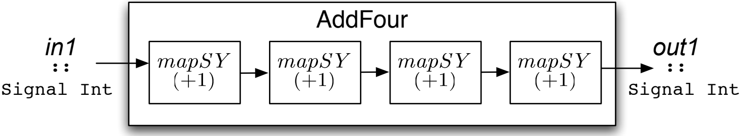


2) Create a system function corresponding to the unit adder

```
addOneProc :: Signal Int32 -> Signal Int32
addOneProc = mapSY addOnef
```

Components

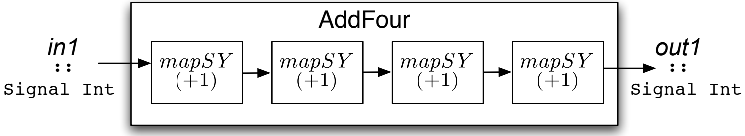
- Similarly to HDLs like VHDL, ForSyDe has support for hierarchical design through components.
- Let's see a simple example. Design a serial adder using components in 5 simple steps.



3) Subsystem definition associated to the unit adder

```
addOneSysDef :: SysDef (Signal Int32 -> Signal Int32)
addOneSysDef = $(newSysDef 'addOneProc ["in1"] ["out1"])
```

- Similarly to HDLs like VHDL, ForSyDe has support for hierarchical design through components.
- Let's see a simple example. Design a serial adder using components in 5 simple steps.

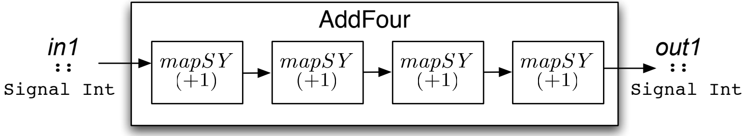


4) Create the main system function

```
addFour :: Signal Int32 -> Signal Int32
addFour = $(instantiate "addOne3" 'addOneSysDef) .
          $(instantiate "addOne2" 'addOneSysDef) .
          $(instantiate "addOne1" 'addOneSysDef) .
          $(instantiate "addOne0" 'addOneSysDef)
```

Components

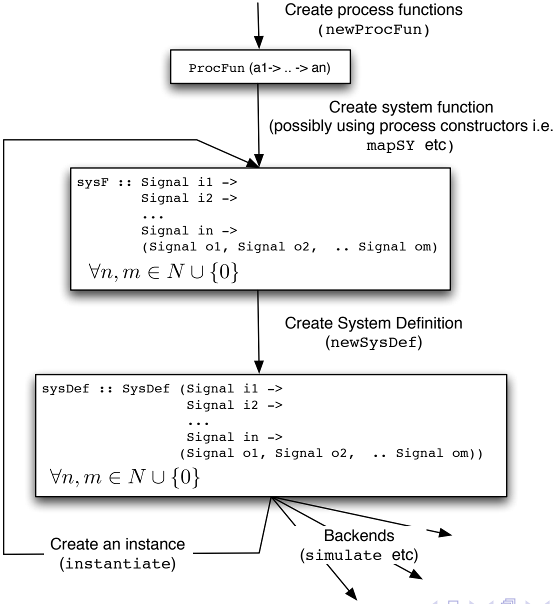
- Similarly to HDLs like VHDL, ForSyDe has support for hierarchical design through components.
- Let's see a simple example. Design a serial adder using components in 5 simple steps.



5) Finally, build the main system definition

```
addFourSys :: SysDef (Signal Int32 -> Signal Int32)
addFourSys = $(newSysDef 'addFour ["in1"] ["out1"])
```

Specification Level Design Flow Using Components



Further Reading

Ingo Sander.
System Modeling and Design Refinement in ForSyDe.
PhD thesis, Royal Institute of Technology, Sweden, 2003.

Alfonso Acosta.
Hardware synthesis in ForSyDe: The design and implementation of a Haskell-embedded ForSyDe-to-VHDL compiler.
Master's thesis, Royal Institute of Technology, Sweden, 2007.

Koen Claessen and David Sands.
Observable Sharing for functional circuit description.
In *Asian Computing Science Conference*, pages 62–73, 1999.

John T. O'Donnell.
Embedding a Hardware Description Language in Template Haskell.
In *Domain-Specific Program Generation*, pages 143–164, 2003.