

A Formalised Framework for Incremental Modelling of On-Chip Communication

Peter Böhm

University of Oxford
Computing Laboratory

Designing Correct Circuits, March 2010

Goal

- ▶ Design of verified high-performance, on-chip communication protocols

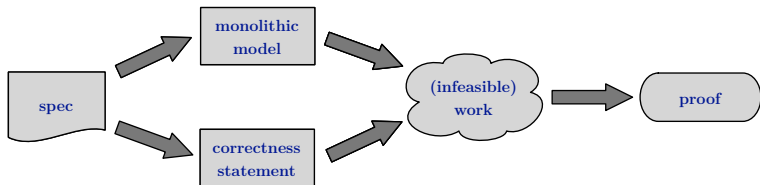
Problem

- ▶ **Communication protocols** traditionally hard to verify
- ▶ **On-chip**: increasing complexity (many-core architectures, System-on-Chips)
- ▶ **High-performance**: hard, advanced features to meet performance demands
- ▶ **Fundamental**: correct execution relies on correct data exchange

Need for functional verification

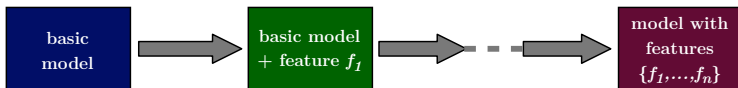
Traditional verification approach usually infeasible

- ▶ Complex, **monolithic model**
 - ▶ High-performance features
 - ▶ Distributed, concurrent communication system
- ▶ Hard **post-hoc verification** process
 - ▶ large state space
 - ▶ complex correctness property (features)



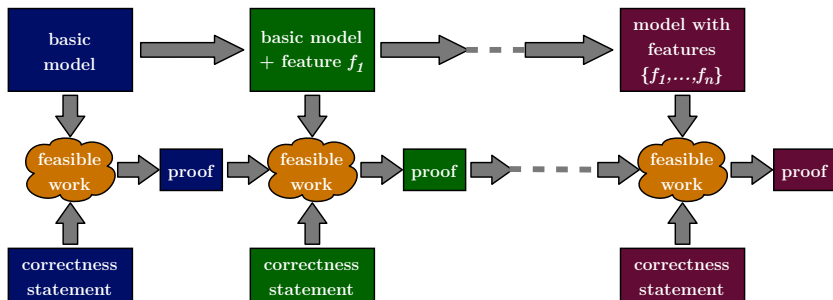
Idea: use **sequence of incremental modelling steps** to replace monolithic model

- ▶ **Basic model** with core functionality
- ▶ Incrementally add **features** in a structured, well defined way
- ▶ Features modelled independently using **transformations**
- ▶ **Complexity encapsulated**



Idea: **spread verification** over modelling process

- ▶ Basic model verified using traditional approach (feasible due to model size)
- ▶ Show correctness of every modelling step
- ▶ **Leverage previous correctness properties**
- ▶ **Reuse** previously proven properties (lemmas)





How to create a sequence of incremental models?

- ▶ Mathematical framework for incremental modelling
 - ▶ Modelling approach
 - ▶ Generic composition operators
 - ▶ Specific transformations
- ▶ Formalisation in Isabelle/HOL

How to apply the methodology?

- ▶ Overview of case study: PCI Express Transaction Layer
 - ▶ Basic model
 - ▶ Specific transformations

General Idea



Model communications system components as **state machines**

- ▶ **Mealy machines**
- ▶ Define a **generic structure** for state space, input and output sets

Extend state machines with **model of communication and composition**

- ▶ Introduce an **interface standard** for the inputs and outputs
- ▶ Provides basis for the model of composition

Define **generic transformations** using composition operators



Definition (Mealy Machine)

A state machine is given by a 6-tuple $(S, I, O, s_0, \delta, \omega)$ where the components are given by

- ▶ S, I, O are the sets for state space, the inputs, and the outputs, respectively.
- ▶ $s_0 \in S$ is the initial state.
- ▶ $\delta : S \times I \rightarrow S$ is the step function of the automaton, thus $\delta(s, i)$ is the next configuration of the automaton with the configuration s and the input assignment i .
- ▶ $\omega : S \times I \rightarrow O$ is the output function of the automaton, thus $\omega(s, i)$ is the assignment of the output values if the state machine is in configuration s and the input assignment is i .

Sets of labelled tuples: structure the sets of a state machine

- ▶ Sets are **collections of tuples**
- ▶ Provide **names for tuple components** to access specific components

Example (Record)

Assume $R = \langle a \in \mathbb{B}, b \in \mathbb{B} \rangle$ with $\mathbb{B} = \{T, F\}$.

Then,

- ▶ $\mathcal{R} = \mathbb{B}^2$
- ▶ $a : \mathbb{B}^2 \rightarrow \mathbb{B}$ with $a((x, y)) = x$
- ▶ $b : \mathbb{B}^2 \rightarrow \mathbb{B}$ with $b((x, y)) = y$
- ▶ Given $r = \langle a = F, b = T \rangle \in R$, then $r.a = a((F, T)) = F$



Definition (Record)

A record set $R = \langle l_0 \in \mathcal{S}_0, \dots, l_i \in \mathcal{S}_i, \dots, l_n \in \mathcal{S}_n \rangle$ of $(n+1)$ -tuples is a set \mathcal{R} with

$$\mathcal{R} = \{(s_0, \dots, s_i, \dots, s_n) \mid \forall j \in [0, n]. s_j \in \mathcal{S}_j\} = \mathcal{S}_0 \times \dots \times \mathcal{S}_i \times \dots \times \mathcal{S}_n$$

together with labelling functions $l_i : \mathcal{R} \rightarrow \mathcal{S}_i$ for each tuple component:

$$l_i((s_0, \dots, s_i, \dots, s_n)) = s_i$$

Notation:

- ▶ A record instance $r \in R$ is given by $\langle l_0 = s_0, \dots, l_i = s_i, \dots, l_n = s_n \rangle$ with $s_j \in \mathcal{S}_j$ for $j \in [0, n]$.
- ▶ Given a record instance $r \in R$, we write $r.l_i \in \mathcal{S}_i$ for $l_i(r)$.

Goal

- ▶ Model **communication between network components** via channels.
- ▶ Specify **operators for composing state machines**.

Uni-directional communication



$$inp_d.y = out_s.x = (\omega_s(s_s, inp_s)).x$$

- ▶ Define communication as a **global function** over a set of state machines
- ▶ **Component aggregates** of input and output records

Component Aggregates of Records

Example

Assume $\mathcal{RS} = \{R_0, \dots, R_n\}$ with $R_i = \langle a \in \mathbb{B}, b \in \mathbb{B} \rangle$ and $n = 2$, then

$$\text{Agg}(\mathcal{RS}) = \{r_0.a, r_0.b, r_1.a, r_1.b, r_2.a, r_2.b\}$$

Definition (Component Aggregate of Records)

Given a set of records $\mathcal{RS} = \{R_0, \dots, R_n\}$, we define the component aggregate of \mathcal{RS} as $\text{Agg}(\mathcal{RS})$ with

$$\text{Agg}(\mathcal{RS}) = \{r_i.x \mid r_i \in \mathcal{R}_i \wedge (\exists j. x = l_j)\}$$



Communication among a set of state machines

- ▶ **Global function** mapping inputs to outputs.
- ▶ **Semantics:** every *data element* produced by the output is communicated to the input given by the function.
- ▶ An external input of a state machine gets defined by the output function of another state machine.

Definition

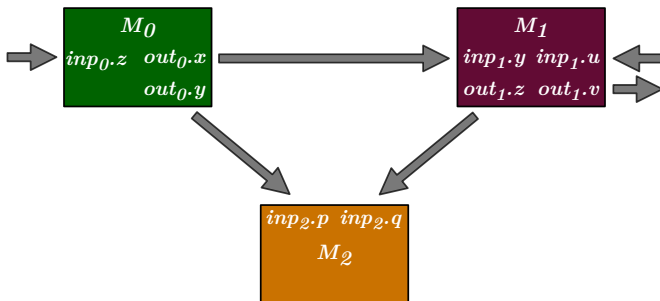
Given a set of state machines $\mathcal{M} = \{M_0, \dots, M_n\}$ with input records I_i and output records O_i . We define the communication as a partial function $\text{com}_{\mathcal{M}} : \text{Agg}(\{I_i \mid i \in [0, n]\}) \rightarrow \text{Agg}(\{O_i \mid i \in [0, n]\})$ such that

$$\text{com}_{\mathcal{M}}(\text{inp}_i.y) = \begin{cases} \text{out}_j.x & : \text{output } x \text{ of } M_j \text{ is send to } M_i \text{ using input } y \\ \text{undefined} & : \textit{otherwise} \end{cases}$$

Global Communication Function: Example

Example

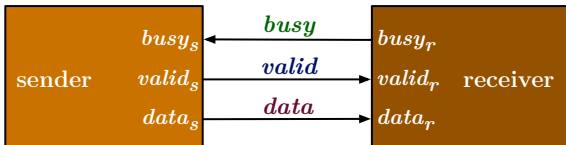
- ▶ $\mathcal{M} = \{M_0, M_1, M_2\}$, $M_i = (S_i, I_i, O_i, s0_i, \delta_i, \omega_i)$
- ▶ $\text{com}_{\mathcal{M}} = \{(inp_1.y, out_0.x), (inp_2.p, out_0.y), (inp_2.q, out_1.z)\}$



Interface Convention

Simple handshake

- ▶ Introduce **standard interface** specification between components as basis for **composition operators**
- ▶ $busy \in \mathbb{B}$, $valid \in \mathbb{B}$, $data \in \mathcal{D}$ where \mathcal{D} is the set of data elements to be communicated.



Semantics

- ▶ If sender wants to send data element x : $valid_s = \text{T}$ and $data_s = x$
- ▶ If $busy_r = \text{F}$: receiver samples data in the same time step.
- ▶ If $busy_r = \text{T}$: receiver is busy and cannot sample data.
 Sender has to provide data until $busy_r = \text{F}$, or data is not communicated.

A Generic Buffer

- ▶ Use **polymorphism** to define generic constructs
- ▶ Use the **option data type** for the data signal to formalise valid and data signals. Then the valid signal corresponds to $data = \text{Some } x$

Definition ((α) buffer of finite size)

A generic buffer of finite size $l \in \mathbb{N}$ is given by the state machine $(S, I, O, s0, \delta, \omega)$ with

$$S = (\mid data \in (\alpha)\text{list}, length \in \mathbb{N} \mid)$$

$$I = (\mid busy \in \mathbb{B}, data \in \alpha \text{ option} \mid)$$

$$O = (\mid busy \in \mathbb{B}, data \in \alpha \text{ option} \mid)$$

$$s0 = (\mid data = \text{Nil}, length = l \mid)$$

$$\delta = \lambda s \in S. \lambda i \in I. \text{let}$$

$$s' = \text{if } \neg(i.\text{busy} \vee s.\text{data} = \text{Nil}) \text{ then } s' = (\text{tail } s.\text{data}) \text{ else } s' = s.\text{data}$$

$$s'' = \text{if } (i.\text{data} = \text{Some } x) \text{ then } s'' = s'@[x] \text{ else } s'' = s'$$

$$\text{in } (\mid data = s'', length = s.length \mid)$$

$$\omega = \lambda s \in S. \lambda i \in I. \text{let}$$

$$\text{out} = \text{if } \neg(s.\text{data} = \text{Nil}) \text{ then } \text{Some } (\text{head } s.\text{data}) \text{ else } \text{None}$$

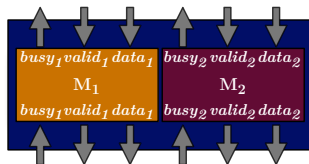
$$\text{in } (\mid busy = (\text{length } s.\text{data} = l), data = \text{out} \mid)$$

Parallel and Sequential Composition

- ▶ Standard (straightforward) composition operators
- ▶ Mainly used to **compose stack layers**

Parallel Composition

- ▶ **Goal:** Execute two state machines M_1, M_2 in parallel
- ▶ All inputs and outputs are inputs and outputs of the composed state machine.



Definition (Parallel Composition Operator)

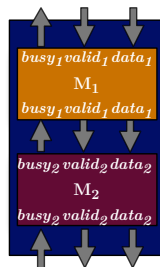
The parallel composition $M_1 \text{ par } M_2$ of state machines M_1 and M_2 with $M_i = (S_i, I_i, O_i, s\theta_i, \delta_i, \omega_i)$ is defined as $(S, I, O, s\theta, \delta, \omega)$ with

$$\begin{aligned}
 (S, I, O) &= (\langle m_1 \in S_1, m_2 \in S_2 \rangle, \langle m_1 \in I_1, m_2 \in I_2 \rangle, \langle m_1 \in O_1, m_2 \in O_2 \rangle) \\
 s\theta &= \langle m_1 = s\theta_1, m_2 = s\theta_2 \rangle \\
 \delta &= \lambda s \in S. \lambda i \in I. \langle m_1 = \delta_1 \text{ s.m}_1 \text{ i.m}_1, m_2 = \delta_2 \text{ s.m}_2 \text{ i.m}_2 \rangle \\
 \omega &= \lambda s \in S. \lambda i \in I. \langle m_1 = \omega_1 \text{ s.m}_1 \text{ i.m}_1, m_2 = \omega_2 \text{ s.m}_2 \text{ i.m}_2 \rangle
 \end{aligned}$$

Parallel and Sequential Composition

Sequential Composition

- ▶ **Goal:** Execute two state machines M_1, M_2 **sequentially**
- ▶ Data outputs of M_1 are connected to the inputs of M_2
- ▶ Remaining inputs and outputs are inputs and outputs of the composed state machine



Parallel and Sequential Composition

Definition (Sequential Composition Operator)

The sequential composition $M_1 \text{seq} M_2$ of state machines M_1 and M_2 with $M_i = (S_i, I_i, O_i, s\theta_i, \delta_i, \omega_i)$ is defined as $(S, I, O, s\theta, \delta, \omega)$ with

$$\begin{aligned}
 (S, I, O) &= (\langle m_1 \in S_1, m_2 \in S_2 \rangle, I_1, O_2) \\
 s\theta &= \langle m_1 = s\theta_1, m_2 = s\theta_2 \rangle \\
 \delta &= \lambda s \in S. \lambda i \in I. \langle m_1 = \delta_1 \text{ s.m}_1 \text{ int}_1, m_2 = \delta_2 \text{ s.m}_2 \text{ int}_2 \rangle \\
 \omega &= \lambda s \in S. \lambda i \in I. \langle m_1 = \omega_1 \text{ s.m}_1 \text{ int}_1, m_2 = \omega_2 \text{ s.m}_2 \text{ int}_2 \rangle
 \end{aligned}$$

where

$$\begin{aligned}
 \text{int}_1 &= \langle \text{busy} = (\omega_2 \text{ s.m}_2 \langle \text{busy} = i.\text{busy}, \text{valid} = \text{F}, \text{data} = x \rangle).\text{busy}, \\
 &\quad \text{valid} = i.\text{valid}, \text{data} = i.\text{data} \rangle \text{ for some } x \\
 \text{int}_2 &= \langle \text{busy} = i.\text{busy}, \text{valid} = (\omega_1 m_1 \text{ int}_1).\text{valid}, \text{data} = (\omega_1 m_1 \text{ int}_1).\text{data} \rangle
 \end{aligned}$$

Note:

- ▶ Definition relies on the assumption that the busy output signal is independent from the valid and data input signals.
- ▶ Assumption needs to be discharged when sequential composition is used.

Goal: control and/or modify data output of a state machine.

▶ **State space**

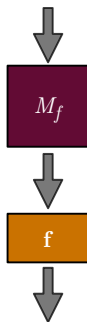
$S = (\{m \in M_f, e \in \mathcal{E}\})$ where \mathcal{E} is a state space extension specific to the function f .

▶ **Input/Output domain**

$I = I_f$, $O = (\{busy \in \mathbb{B}, valid \in \mathbb{B}, data \in \mathcal{F}\})$
where \mathcal{F} is the range of the function f .

▶ **Combinatorial function** $f : \mathcal{D} \rightarrow \mathcal{F}$ where \mathcal{D} is the data output range of M_f .

- ▶ Combinatorial in the sense that data elements are not *stored*.
- ▶ Step function for f to update state space element e .
- ▶ Output function for f that depends on e and the input signal, i. e. the output signals of M_f .



Generic Multiplex/Arbitrate Composition

Goal: controlled, parallel execution of $n + 1$ state machines M_i while maintaining the input and output interface.

► **State space**

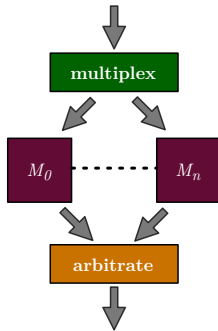
$S = (\{m_0 \in S_0, \dots, m_n \in S_n, e \in \mathcal{E}\})$ where \mathcal{E} is a state space extension specific to a concrete instance of the operator.

► **Input domain**

$I = (\{busy \in \mathbb{B}, valid \in \mathbb{B}, data \in \bigcup_i \mathcal{D}_i\})$ where \mathcal{D}_i is the data domain of M_i . **Output domain** is defined analogously.

► **Multiplex relation** $mux \subseteq (S \times I) \times [0, n]$ to select the internal component(s) given input signal values.

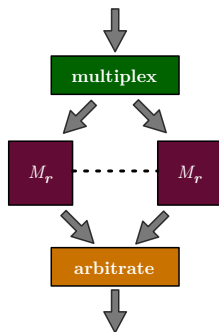
► **Arbitrate function** $arb : (S \times I) \rightarrow [0, n]$ to select the component that outputs data.



Replicate Composition

Goal: controlled, parallel execution of $n + 1$ **copies** of a state machine M_r .

- ▶ Similar to the generic multiplex/arbitrate, but more restrictive
- ▶ Advantage: more correctness results
- ▶ **State space**
 $S = (\{m_0 \in S_r, \dots, m_n \in S_r, e \in \mathcal{E}\})$
- ▶ **Input/Output domain**
 $I = I_r, O = O_r$
- ▶ **Multiplex function** $max : (S \times I) \rightarrow [0, n]$
 (instead of relation)
- ▶ **Arbitrate function** $arb : (S \times I) \rightarrow [0, n]$
 analogous to multiplex/arbitrate composition



- ▶ Argue about behaviour over time
- ▶ Intuitive, standard definition
- ▶ Abstract, discrete time domain: \mathbb{N}

Definition (Signal)

A signal sig is a function from time \mathbb{N} to a data domain \mathcal{D} . We write sig^t for $sig(t)$.

Definition (Execution and Output Trace)

Given a state machine $M = (S, I, O, s0, \delta, \omega)$ and input values $i^t \in I$ for $t \in \mathbb{N}$, we define the execution trace $trc_M : \mathbb{N} \rightarrow S$ and the output trace $out_M : \mathbb{N} \rightarrow O$ as

$$\begin{aligned}
 trc_M^t &= \begin{cases} s0 & : t = 0 \\ \delta trc_M^{t-1} i^{t-1} & : otherwise \end{cases} \\
 out_M^t &= \omega trc_M^t i^t
 \end{aligned}$$

Correctness:

- ▶ Functional correctness (no data loss or modification)
- ▶ No reordering
- ▶ Liveness

Environment assumption:

- ▶ *busy* input not constantly active

Lemma (Correctness of the Buffer FSM)

Given input signals $i^t \in I$, a generic buffer (α)buffer satisfies that

$$\forall x \in \alpha. \neg i.\text{busy}^t \wedge (i.\text{data}^t = \text{Some } x) \implies \exists k. (\text{out}_M^{t+k} = \text{Some } x)$$

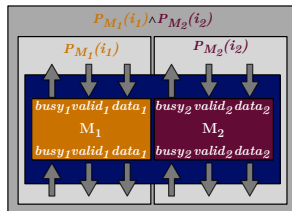
Note

- ▶ Analogous lemma with $x_1, x_2 \in \alpha$ shows in-order property.
- ▶ Easy lemma to show that data outputs independent of busy input.

Basic Compositions

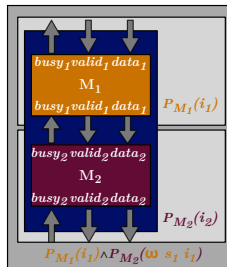
Parallel Composition

- ▶ Correctness properties of the components are maintained.
- ▶ Satisfies conjunction of the individual correctness properties.
- ▶ Environment assumptions of both state machines have to be satisfied.



Sequential Composition

- ▶ Satisfies conjunction of the correctness with the respective substitutions in P_{M_2} using ω_1 .
- ▶ Analogously for the *busy* input of M_1 (definition of sequential composition)
- ▶ Data output of M_1 has to satisfy the environment assumptions of M_2 and vice versa for the *busy* input.

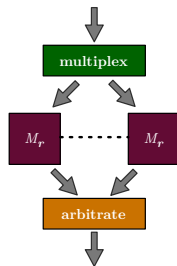


Replication Operator

Idea: Push correctness from inner components to system

Assumptions:

- ▶ M_r is **correct and ensures liveness**
- ▶ The multiplex function is **correct for valid inputs**
- ▶ The arbitration function is **fair** with respect to an active valid signal from some M_r



Theorem (Functional Correctness and Liveness)

The replication operator preserves the functional correctness and the liveness of M_r given the above assumptions.

Protocol characteristics

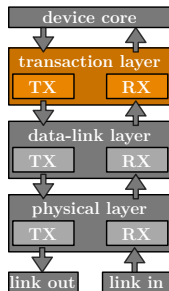
- ▶ Point-to-point, packet-based communication
- ▶ Protocol stack layers: Transaction, data-Link, physical Layer
- ▶ Each layer: transmit (TX) and receive part (RX)

Memocode'09: Derivation of transaction layer

- ▶ Focus on hard transaction layer parts
flow control, packet reordering, virtual channels
- ▶ Transformation-based modelling approach
- ▶ Formalization in Isabelle/HOL

Here: Summary of

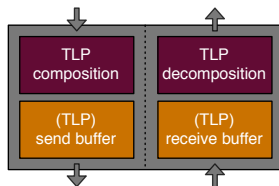
- ▶ Basic model
- ▶ Flow control



Data units: transaction layer packets (TLPs)

Model

- ▶ TLP composition/decomposition
- ▶ Send/receive buffers



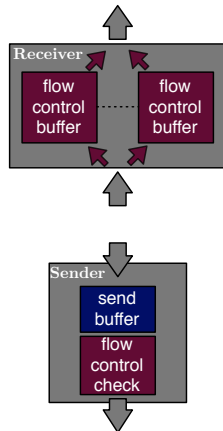
Correctness

- ▶ TLP composition/decomposition (only combinatorial, easy)
- ▶ Apply correctness of generic buffer
 - ▶ Liveness
 - ▶ Ordering (no overtaking or packet loss)
 - ▶ Correct busy signal
- ▶ Sequential composition of TX, channel, and RX

Goal: Sender checks locally if receiver has enough buffer space.

Principle

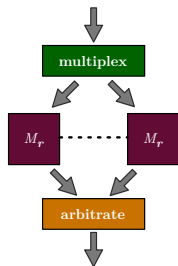
- ▶ Credit-based (header 1 credit, dw 1 credit)
- ▶ **Receiver:** Flow control buffers
 - ▶ For each message type (posted, non-posted, completion)
 - ▶ Header and payload (not every packet as payload)
 - ▶ Frequent updates to link neighbour
- ▶ **Sender:** Checks if space is available
 - ▶ Maintains available space counters
 - ▶ Checks before message transmission



Flow Control - A Specific Transformation

Receiver: Instantiate replication operator

- ▶ $n = 3$ with $(TLP, timestamp)$ flow control buffer
- ▶ Multiplex function is TLP to $[0 : 2]$ plus add time stamp
- ▶ Arbitrate function is n such that $timestamp(n) < timestamp(m)$ for all $m \neq n$



Flow control buffer: Instantiate multiplex/arbitrate operator

- ▶ $n = 2$ with $(TLPHeader, timestamp)$ and $(TLPData)$ data buffer
- ▶ Multiplex relation is $\{0\}$ if TLP has no data and $\{0, 1\}$ if TLP has data
- ▶ Arbitrate relation analogous to multiplex relation with respect to busy input

Sender: Instantiate combinatorial function operator

- ▶ Combinatorial function is counter test; raise *busy* if there is not enough space

PCI Express Summary



- ▶ **Industrial-sized high-performance** communication protocol
- ▶ **Incremental modelling** of large parts of the transaction layer and data-link layer
- ▶ **Independent specification** of complex features
- ▶ **Transaction layer**
 - ▶ Flow control
 - ▶ TLP reordering
 - ▶ Packet priorities using virtual channels
- ▶ **Data-link layer**
 - ▶ Data-link layer packet arbitration
 - ▶ ACK/NAK protocol
 - ▶ CRC check
- ▶ Case study results published in MEMOCODE'09 and HFL'09

New methodology for an incremental modelling and verification process

- ▶ Control the model complexity by adding features incrementally
- ▶ Formalised framework with correctness results for the generic constructs
- ▶ Generalised design principle for transformations using composition operators
- ▶ HOL as design/modelling language

Long-term aim

- ▶ Increase efficiency of the model building process
- ▶ Model with significant merits against ad-hoc models
 - ▶ Functional verified
 - ▶ Independent from implementation or design architecture
 - ▶ Long-term reference model

Theorem prover

- ▶ Reduce or eliminate manual theorem proving
- ▶ Ideally modelling tool with knowledge management features

PCI Express

- ▶ Support for power management and interrupts
- ▶ Derivation of switches (support for complex topologies)

Design and verification methodology

- ▶ Support for (automatic) refinement steps (data refinement)
- ▶ Integration of automated verification tools (model checking, SMT Solver)
- ▶ Link to HDL? (SystemVerilog)