A Framework for Incremental Modelling and Verification of On-Chip Protocols

Peter Böhm
Oxford University Computing Laboratory
Oxford, OX1 3QD, England
peter.boehm@comlab.ox.ac.uk

***Abstract*—Arguing formally about the correctness of on-chip communication protocols is an acknowledged verification challenge. We present a generic framework that tackles this problem using an incremental approach that interleaves model construction and verification.**

Our protocol models are based on abstract state machines formalized in Isabelle/HOL. We provide abstract building blocks and generic composition rules to support incremental addition of protocol features to a parameterized endpoint model. This structured approach controls model complexity. We can refine data structures and develop control independently, to create a concrete instantiation.

To make the verification effort feasible, we combine interactive theorem proving with symbolic model checking using NuSMV. The theorem prover is used to reason about generic correctness properties of the abstract models given some local assumptions. We can use model checking to discharge these assumptions for a specific instantiation. We show the utility and breadth of the framework by sketching two case studies: modelling a bus protocol, and modelling the PCI Express point-to-point protocol.

I. Introduction

Formal verification of high-performance on-chip communication protocols is widely acknowledged to be hard. Modern multi- or many-core architectures require highly complex protocols to handle the performance bottleneck due to communication. These protocols implement sophisticated features to provide the needed performance.

Traditionally, monolithic models are created and proven correct using post-hoc verification. Given the complexity of the features and the size of the distributed system, this approach became often infeasible in time and effort.

We propose a new methodology based on incremental modelling and step-wise verification to tackle this challenge. The idea is to encapsulate the complexity of the features into independent modelling steps and add these features incrementally to the model, starting with a very simple model. At the same time, we reduce the verification effort with two main strategies: first the verification process can be spread over the modelling process such that in each step we only need to verify the parts added to the model that implement the new feature. Second we use generic building blocks for which we have shown correctness results. The verification can be restricted to discharging local assumptions on the building blocks.

In previous contributions [1], [2], we have illustrated the approach on two case studies. To explore the utility and breadth of the approach, we chose two rather different examples: first the ARM AMBA Advanced High-performance Bus (AHB) protocol [3], an arbiter-based master-slave bus protocol. As a second protocol, we picked the PCI Express protocol [4], a modern point-to-point protocol. We will briefly summarize the application of the general framework to these protocols in Section VI.

The development of the framework was driven by these case studies. After completing the work on both protocols, we were able to create a protocol independent formalization of a mature framework which we present here. All the models have been formalized in higher order logic using the Isabelle theorem prover [5].

For the case studies, we realized the modelling and verification idea using interactive theorem proving only. However, our final aim was to reduce the theorem proving part to further increase the feasibility of the approach. Here, we show how to integrate automatic tools into the verification workflow. On the one hand, we use automatic theorem provers for subgoals in first-order logic using the sledgehammer interface of Isabelle/HOL (Isabelle 2009-1). Sledgehammer invokes the provers E, SPASS, and Vampire and, if successful, returns a proof script for the Metis theorem prover.

On the other hand, we integrate the NuSMV symbolic model checker in the workflow. To use the model checker, we adapt the oracle-based IHaVeIt interface [6] to Isabelle 2009. We can invoke NuSMV to prove LTL and CTL formulas from Isabelle. This is especially useful when we instantiate generic composition operators and need to discharge local assumptions, such as fairness constraints. We will detail the integration of automated tools in Section V.

Our main contributions can be summarized as follows:

- A framework based on (abstract) state machines for modelling and verification of on-chip protocols.
- Generic building blocks and composition rules to create models incrementally covering basic components such as buffers to specialised composition schemes.
- A verification methodology that handles the complexity by restricting the effort to local constraints and global, generic correctness results.
- Integration of the NuSMV model checker in the verification process to further reduce the verification effort with focus on automatically discharging the local constraints.

These contributions result in a promising prototype system that has been successfully applied to a variety of protocol features.

A. Related Work

Modelling systems with automata in general is a well studied field and can be found, for example, in the widely-cited book by Robert Kurshan [7]. Formalizing state machines in Isabelle/HOL goes back to at least Nipkow and Slind [8]. They formalized I/O automata and developed a meta-theory to represent them as objects in the logic. We restrict our state machine framework to a simpler formalization specialised to our requirements. Formal verification of protocols using I/O automata in theorem provers has a long history, e.g. [9], [10]. Our goal is not to provide yet another specific protocol verification using I/O automata and a theorem prover, but the formalization of a methodology.

Suhaib *et al.* [11] propose an incremental methodology for developing formal models called XFM. An extendable set of LTL properties is used to incrementally create a model that satisfies the set of properties. Their approach focuses on building prescriptive formal models that capture the behaviour of natural language specifications. Our methodology tries to capture specific features in independent models.

Another related approach is the B Method [12], an event-based method for a refinement-based specification, design and implementation of software components. Abrial *et al.* [13] apply the method to the incremental development of the IEEE 1394 tree identify protocol. Cansell *et al.* present an incremental proof of the producer/consumer property for the PCI protocol using the approach. Besides being tailored to software and being event-based, our approach is not only restricted to refinement steps.

Schmaltz *et al.* [14] present a generic network on chip model as a framework for correct on-chip communication. They identify key constraints on architectures and show protocol correctness given these constraints. However, their work focuses on the topologies in general, whereas this work aims at the verification of sophisticated endpoints.

Chen *et al.* [15] propose a modular, refinement based approach to verify transaction-based hardware implementations against their specification models and illustrate their methodology using a cache coherency protocol. Their approach is tailored to a different application area: verifying implementations against specifications.

Müffke [16] presents a framework for the design of communication protocols. He provides a dataflow-based language for protocol specification and decomposition rules for interface generation relating dataflow algebra and process algebra. Aside from noting that correct and verified protocol design is still an unsolved problem, Müffke does not address the verification aspect in general.

General hardware verification based on refinement checking or simulation relations has a long history. Finn and Fourman [17] present the toolset LAMBDA, a refinement based general-purpose design assistant using mathematical logic to represent and manipulate system behaviour. Abadi and Lamport [18] show the existence of refinement mappings in their widely-cited article. McMillan [19] proposes a compositional rule for hardware verification based on local refinements which can be efficiently model checked.

The combination of Isabelle/HOL and NuSMV using the IHaVeIt tool has been applied to a variety of hardware verification instances. Schmaltz [20] applies it to the area of clock domain crossing and the time-triggered hardware implementing it. Alkassar *et al.* [21] use the tool to show the correctness of a fault-tolerant real-time scheduler and its hardware implementation. In both cases, the authors apply a similar strategy: they use theorem proving to argue about real-time, asynchronous properties of the system, and the model checker to prove properties of finite state machines which are used to model the hardware implementation. A more general overview of *hybrid verification approaches* can be found in the survey from Bhadra *et al.* [22]

An overview of existing work on specific protocol verification such as PCI Express or the AMBA protocol can be found in our previous contributions covering the case studies [1], [2]. As this work focuses on the protocol independent, general methodology, we omit the protocol specific work here.

II. Basics

A. Notation

To represent data, we often use the option datatype and records which we introduce in the following. Moreover, we define discrete time signals and introduce correctness properties.

a) Option Type: To specify a possibly undefined value, we use the *option datatype* that is well known from functional programming languages. For an element of type $(\alpha)option$, we write $\text{Some } x$ for $x \in \alpha$ and None for the two constructors. The selection operator $\mathbf{the}$ is used to access a value: $\mathbf{the}(\text{Some } x) = x$ and $\mathbf{the}(\text{None})$ is left unspecified.

b) Records: We use the Isabelle notation for *records*. Let $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be sets and $l_1, \ldots, l_n$ be labels. The record $\mathcal{R} = (\!| \, l_1 : \mathcal{S}_1, \ldots, l_n : \mathcal{S}_n \, |\!)$ yields the set of all tuples $(l_1 = s_1, \ldots, l_n = s_n)$ where $s_i \in \mathcal{S}_i$. For $r \in \mathcal{R}$, we refer to the field l_i with $r.l_i$. An *update* to a field l_i by a value $s_i \in \mathcal{S}_i$ is denoted $r(\!| \, l_i := s_i \, |\!)$. If the context is obvious, we write l_i for $r.l_i$. For $i \in [1, n]$, the domain of field l_i is given by $\mathbf{dom}(l_i, \mathcal{R}) = \mathcal{S}_i$. For a label l, we define the element operator $l \tilde{\in} R$ as $\exists j \in [1, n].\ l = l_j$.

Given the usual Cartesian product for sets, $\prod_{i \in [1,n]} \mathcal{S}_i = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$, we define an analogous operation for records: $\widetilde{\prod}_{i \in [1,n], \mathcal{L}} \mathcal{R}_i = (\!| \, l_1 : \mathcal{R}_1, \ldots, l_n : \mathcal{R}_n \, |\!)$ where $\mathcal{L} : [1 : n] \to \{l_i \mid i \in [1, n]\}$ is a *labelling function* that assigns a label to each index. This labelling function can be given as an explicit function or a set of pairs. We also use this operator for sets as arguments, i. e. to create a record from sets. A disjoint union operator over records is given by the set of record fields: $\widetilde{\biguplus}_{i \in [1,n]} \mathcal{R}_i = \{l_{i,j} \mid l_j \tilde{\in} \mathcal{R}_i\}$. Finally, we define the concatenation of records: $\mathcal{R}_i = (\!| \, l_{i,0} : \mathcal{S}_{i,0}, \ldots, l_{i,m_i} : \mathcal{S}_{i,m_i} \, |\!)$: $\widetilde{\bigodot}_{i \in [0,n]} \mathcal{R}_i = (\!| \, l_{0,0} : \mathcal{S}_{0,0}, l_{0,1} : \mathcal{S}_{0,1}, \ldots, l_{n,m_n} : \mathcal{S}_{n,m_n} \, |\!)$. We use $R_0 \tilde{\uplus} R_1$, $R_0 \tilde{\times}_{\mathcal{L}} R_1$, and $R_0 \tilde{\odot} R_1$ for the binary variants.

c) Signals: A *signal sig* is a function from discrete time to a signal data type α, i.e. $sig : \mathbb{N} \to \alpha$. We denote the value of a signal sig at time t with sig^t.

d) Correctness Properties: In the context of this paper, a *correctness property* P is either a propositional logic formula, an LTL formula, or a CTL formula given a state machine model M, an external input signal i, and a set of assumptions $\mathcal{A}$. If the model and the input signal satisfies the correctness property given the assumptions, we write $\langle M, i\rangle \models_{\mathcal{A}} P$.

B. Communicating Abstract State Machines

We use standard Mealy machines to represent components in the framework. A state machine is specified by an *initial state*, a *transition function*, and an *output function* together with sets for the *state space*, the *input space*, and the *output space*.

Definition 1 (Mealy Machine): A Mealy machine is given by a 6-tuple $(S, I, O, s0, \delta, \omega)$ with domains S, I, O, initial state $s0 \in S$, transition function $\delta : S \times I \to S$, and output function $\omega : S \times I \to O$. We denote the next state $s' = \delta(s, i) \in S$ and the current output is $\omega(s, i) \in O$.

If not stated otherwise, we assume that the domain spaces are given as records. Given a state machine and an input signal, we can define the *execution trace* and the *output signal*.

Definition 2 (Execution Trace and Output Signal): Given a state machine $M = (S, I, O, s0, \delta, \omega)$ and an input signal $in : \mathbb{N} \to I$, we define the execution trace τ as $\tau^0_{M,in} = s0$ and $\tau^t_{M,in} = \delta(\tau^{t-1}_{M,in}, in^{t-1})$ for $t > 0$. The output signal, $out_{M,in} : \mathbb{N} \to O$, is given by $out^t_{M,in} = \omega(\tau^t_{M,in}, in^t)$.

To define composition operators in Section IV, we introduce a model of synchronous communication among state machines. We model uni-directional communication from a *source* to a *destination* by connecting an output of the source to an input of the destination. This 'connection' is modelled by defining the input component using the output function of source state machine. To illustrate the general approach, assume we want to model a communication from output $x \tilde{\in} O_s$ to input $y \tilde{\in} I_d$. Given an input signal $i^t_d \in I_d$, we use the following definition for its record component y instead of considering it as an environment input: $i^t_d.y = (\omega_s(\tau^t_{M_s,in_s}, in^t_s)).x$. We can generalize this approach by introducing a global communication function for a set of abstract state machines.

Definition 3 (Communication Function): Given a set of state machines $\mathcal{M} = \{M_0, \ldots, M_n\}$. We define communication as a partial function $\mathbf{com}_{\mathcal{M}} : \widetilde{\biguplus}_i I_i \to \widetilde{\biguplus}_i O_i$ such that $\mathbf{com}_{\mathcal{M}}(y_i) = x_j$ if output x of M_j is connected to y of M_i and undef otherwise. We call an input y of M_i *external* with respect to $\mathcal{M}$ iff $\mathbf{com}_{\mathcal{M}}(y_i) = \text{undef}$ and *internal* otherwise.

C. Standard Interface

We use a simple *handshake* protocol to realise a uni-directional communication between two state machines using three signals: a *valid* and a *data* signal provided by the sender, and a *busy* from the receiver. The basic idea is that a sender provides data on the data signal and raises the valid signal to indicate so. If the receiver's busy signal is low, it is an acknowledgement that the receiver samples the data in the same time step. If the busy signal is active, the receiver cannot sample the data yet and the sender has to keep its signals stable. We refer to the three signals with $b^t \in \mathbb{B}$, $v^t \in \mathbb{B}$, and $d^t \in \mathcal{D}$ where $\mathcal{D}$ is the set of data elements to be communicated. We use a suffix o to refer to an output, and a suffix i for an input.

We formalise the protocol in terms of two assumptions: one that defines *valid outputs* provided by a sender and one that specifies the *correct sampling* behaviour of a receiver.

Assumption 1 (valid outputs): M provides *valid output* at time t iff $vo^t \implies (do^t = x) \wedge \big(bi^t \implies vo^{t+1} \wedge (do^{t+1} = x)\big)$ for some data element $x \in O$.

Assumption 2 (correct sampling): M has to sample input data $x = di^t$ at time t iff $vi^t \wedge \neg bo^t$.

We use the option datatype to model the data signal and omit the valid signal. The interface consists of two signals: a busy signal $b^t \in \mathbb{B}$ and a data signal $d^t \in \mathcal{D}$ *option*. The valid signal can be obtained by $v^t \equiv (d^t \neq \text{None})$. Next, we generalize the concept to specify the input and output records of a state machine implementing n *input interfaces*—the data signal is an input—and m output interfaces—the data signal is an output. We use the following labelling convention for the signal names: the k-th data input is di_k together with the k-th busy output bo_k; analogously for the output interface. Thus, we restrict the input and output records to the following generalised constructs:

$$
\begin{aligned}
I &= \Big(\widetilde{\prod}_{m,\rho_{bi,m}}\mathbb{B}\Big) \mathbin{\tilde{\odot}} \Big(\widetilde{\prod}_{i\in[1,n],\rho_{di,n}}\mathcal{D}_i \ option\Big) \\
&= \mathcal{BI}^m \mathbin{\tilde{\odot}} \mathcal{DI}^n \\
O &= \Big(\widetilde{\prod}_{n,\rho_{bo,n}}\mathbb{B}\Big) \mathbin{\tilde{\odot}} \Big(\widetilde{\prod}_{j\in[1,m],\rho_{do,m}}\mathcal{D}_j \ option\Big) \\
&= \mathcal{BO}^n \mathbin{\tilde{\odot}} \mathcal{DO}^m
\end{aligned}
$$

where the labelling $\rho_{l,k}$ is given by $\rho_{l,k}(i) = l_i$ for $i \in [1, k]$. Given an element $i \in I$, we use $\mathcal{BI}^m(i)$ to refer to the m busy inputs and $\mathcal{DI}^n(i)$ to refer to the n data inputs. For $o \in O$, we use $\mathcal{BO}^n(o)$ and $\mathcal{DO}^m(o)$.

Since we use Mealy machines to model abstract components, it is possible to create combinatorial loops when we compose state machines using this handshaking protocol. We define two interface properties which prevent this. When we introduce operators for state machine composition, we will use these properties as local constraints.

Assumption 3 (busy-independent data): Given a state machine M and an input signal i. M provides busy-independent data output signals iff ω satisfies for all $k \in [1, m]$: $\omega\ (\tau^t_{M,i}, i^t).do_k = \omega\ (\tau^t_{M,i}, i^t(\!|\, bi_k := \mathsf{T} \,|\!)).do_k$

Assumption 4 (data-independent busy): Given a state machine M and an input signal i. M provides data-independent busy signals iff ω satisfies for all $k \in [1, n]$: $\omega\ (\tau^t_{M,i}, i^t).bo_k = \omega\ (\tau^t_{M,i}, i^t(\!|\, di_k := \text{None} \,|\!)).bo_k$

III. Abstract Components

In this section, we introduce basic building blocks: a polymorphic buffer of finite size for arbitrary data elements that obeys the standard interface, a component for data modification, and one for signal routing.

A. Buffer

We model a buffer using a list and specify the basic operations for buffers or queues: an *enqueue* operation enq to add a new element, a *dequeue* operation deq to remove the oldest element, and a *top* operation top to obtain the oldest element. We use predicates $empty$ and $full$ for corresponding buffer states. Using lists, all these operations are straightforward. We put the buffer in a state machine wrapper implementing the standard interface.

Definition 4 ((α) buffer of finite size): A generic buffer for datatype α and finite size $s \in \mathbb{N}$ is given by:

$$\begin{aligned}
&S = \llparenthesis\, buf : \alpha\,\textbf{list}, size : \mathbb{N} \,\rrparenthesis, \quad s0 = \llparenthesis\, buf = \text{Nil}, size = s \,\rrparenthesis \\
&I = \llparenthesis\, bi : \mathbb{B}, di : \alpha\, option \,\rrparenthesis, \quad O = \llparenthesis\, bo : \mathbb{B}, do : \alpha\, option \,\rrparenthesis \\
&\delta = \lambda s \in S.\ \lambda i \in I.\ \text{let} \\
&\qquad s' = \text{if } \neg(i.bi \vee \textbf{empty}\ s) \text{ then } \textbf{deq}\ s \text{ else } s \\
&\quad \text{in if } (i.di = \text{Some } x \wedge \neg\textbf{full}\ s') \text{ then } \textbf{enq}\ s'\ x \\
&\qquad \text{else } s' \\
&\omega = \lambda s \in S.\ \lambda i \in I.\ \text{let} \\
&\qquad d = \text{if } \neg(i.bi \vee \textbf{empty}\ s) \text{ then Some } (\textbf{top}\ s) \\
&\qquad\quad \text{else None} \\
&\quad \text{in } \llparenthesis\, bo = \textbf{full}\ s,\ do = d \,\rrparenthesis
\end{aligned}$$

We refer to such a buffer with $(\alpha, s)\, Buf$.

B. Data Modification

The data modification component is a minimalistic state machine implementing modifications to the data element. We abstract from the modification and model it as a function $f : S \times \alpha \to \beta$. A typical use of the component is the extension of a data element with a sequence number or a check sum. An optional element opt is added to provide required data; together with an initial state opt^0 and an update function δ_{opt}.

Definition 5 (Data Modification): Given a function $\mathbf{f} : S \times \alpha \to \beta$, a data modification is a simple state machine with:

$$\begin{aligned}
&S = \llparenthesis\, opt : Opt \,\rrparenthesis, \quad s0 = \llparenthesis\, opt = opt^0 \,\rrparenthesis \\
&I = \llparenthesis\, bi : \mathbb{B}, di : \alpha\, option \,\rrparenthesis, \quad O = \llparenthesis\, bo : \mathbb{B}, do : \beta\, option \,\rrparenthesis \\
&\delta = \lambda s \in S.\, \lambda i \in I.\, \llparenthesis\, opt = (\delta_{opt}\ (s, i)) \,\rrparenthesis \\
&\omega = \lambda s \in S.\, \lambda i \in I.\, \text{let} \\
&\qquad d = \text{if } (i.di = \text{Some } x) \text{ then Some } (\mathbf{f}\ (s, x)) \\
&\qquad\quad \text{else None} \\
&\quad \text{in } \llparenthesis\, bo = i.bi, do = d \,\rrparenthesis
\end{aligned}$$

C. Routing

The goal of the routing component is to distribute control or data flow. Typical applications are the arbitration among data elements or the generation of messages while stalling incoming data. We also use an optional state component opt with initial state opt^0 and update function δ_{opt}. The core of the building block are two functions f_b and f_d which represent the modification of the busy and data signals.

Definition 6 (Routing Component): Let $f_b : S \to I \to \widetilde{\prod}_{n,\rho_{bo,n}} \mathbb{B}$ and $f_d : S \to I \to \widetilde{\prod}_{j \in [1,m],\rho_{do,m}} \mathcal{D}_j\ option$ be *routing functions*. A routing component is given by:

$$\begin{aligned}
&S = \llparenthesis\, opt : Opt \,\rrparenthesis, \quad s0 = \llparenthesis\, opt = opt^0 \,\rrparenthesis \\
&I = \left(\widetilde{\prod}_{m,\mathcal{L}_{bi}} \mathbb{B}\right)\ \tilde{\odot}\ \left(\widetilde{\prod}_{i \in [1,n],\mathcal{L}_{di}} \mathcal{D}_i\ option\right) \\
&O = \left(\widetilde{\prod}_{n,\mathcal{L}_{bo}} \mathbb{B}\right)\ \tilde{\odot}\ \left(\widetilde{\prod}_{j \in [1,m],\mathcal{L}_{do}} \mathcal{D}_j\ option\right) \\
&\delta = \lambda s \in S.\, \lambda i \in I.\, \llparenthesis\, opt = (\delta_{opt}\ (s, i)) \,\rrparenthesis \\
&\omega = \lambda s \in S.\, \lambda i \in I.\, (f_b\ s\ i) \tilde{\odot} (f_d\ s\ i)
\end{aligned}$$

IV. Composition of Abstract Components

In this section, we detail ways to compose state machines in order to incrementally build complex systems. We introduce two standard operations: *parallel* and *sequential* composition. Parallel composition can be used to combine send and receive parts of an endpoint model, for example. A typical application for sequential composition is the modelling of interconnects or the composition of stack layer models. An overview of the sequential composition is shown in Fig. 1(a).

Definition 7 (Parallel Composition): The parallel composition $M_1 \mathbf{par} M_2$ with $M_i = (S_i, I_i, O_i, s0_i, \delta_i, \omega_i)$ is given by:

$$\begin{aligned}
&S = S_1 \tilde{\times}_{\mathcal{L}} S_2, \quad s0 = \llparenthesis\, m_1 = s0_1,\ m_2 = s0_2 \,\rrparenthesis \\
&I = I_1 \tilde{\times}_{\mathcal{L}} I_2, \quad O = O_1 \tilde{\times}_{\mathcal{L}} O_2 \\
&\delta = \lambda s.\ \lambda i.\ \llparenthesis\, m_1 = \delta_1\ (s.m_1, i.m_1),\ m_2 = \delta_2\ (s.m_2, i.m_2) \,\rrparenthesis \\
&\omega = \lambda s.\ \lambda i.\ \llparenthesis\, m_1 = \omega_1\ (s.m_1, i.m_1),\ m_2 = \omega_2\ (s.m_2, i.m_2) \,\rrparenthesis
\end{aligned}$$

where $\mathcal{L} = \{(1, m_1), (2, m_2)\}$ is the labelling.
To define the sequential composition in a compact way, we need to define some internal signals. Assume we want to compose M_1 sequentially with M_2, i. e. $M_1\, \mathbf{seq}\, M_2$. As illustrated in Fig. 1(a), the busy signals of M_1 are connected to the busy outputs of M_2, and vice versa for the data signals. Since there is a *cyclical* dependency, we need that either M_1 provides busy-independent data outputs (Assumption 3) or M_2 provides data-independent busy outputs (Assumption 4). We first define the sequential composition in Definition 8 assuming the internal signals $bint$ and $dint$ as depicted. We define these signals in Definition 9.

Definition 8 (Sequential Composition): Given M_1, M_2 with $M_i = (S_i, I_i, O_i, s0_i, \delta_i, \omega_i)$ where $I_1 = \mathcal{BI}^m \tilde{\odot} \mathcal{DI}^n$, $O_1 = \mathcal{BO}^n \tilde{\odot} \mathcal{DO}^m$ and $I_2 = \mathcal{BI}^p \tilde{\odot} \mathcal{DI}^m$, $O_2 = \mathcal{BO}^m \tilde{\odot} \mathcal{DO}^p$. Then, the sequential composition $M_1\, \mathbf{seq}\, M_2$ is defined as:

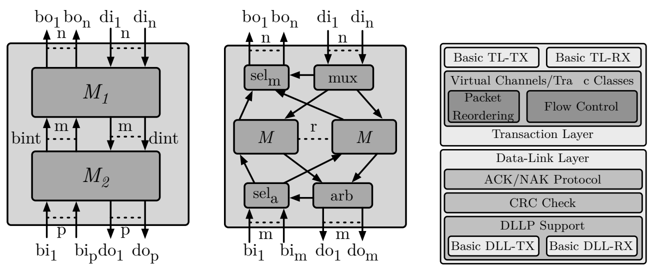
$$\begin{aligned}
&S = S_1 \tilde{\times}_{\mathcal{L}} S_2, \quad s0 = \llparenthesis\, m_1 = s0_1,\ m_2 = s0_2 \,\rrparenthesis \\
&I = \mathcal{BI}^p\ \tilde{\odot}\ \mathcal{DI}^n, \quad O = \mathcal{BO}^n\ \tilde{\odot}\ \mathcal{DO}^p \\
&\delta = \lambda s.\ \lambda i.\ \llparenthesis\, m_1 = \delta_1\ (s.m_1, bint \tilde{\odot} \mathcal{DI}^n(i)), \\
&\qquad m_2 = \delta_2\ (s.m_2, \mathcal{BI}^p(i) \tilde{\odot} dint) \,\rrparenthesis \\
&\omega = \lambda s.\ \lambda i.\ \mathcal{BO}^n(\omega_1\ (s.m_1, bint \tilde{\odot} \mathcal{DI}^n(i)))\ \tilde{\odot} \\
&\qquad\quad \mathcal{DO}^p(\omega_2\ (s.m_2, \mathcal{BI}^p(i) \tilde{\odot} dint))
\end{aligned}$$

where the labelling $\mathcal{L}$ is $\{(1, m1), (2, m2)\}$.

Definition 9 (Internal Signals): Let $M = M_1\, \mathbf{seq}\, M_2$ be the sequential composition of M_1 and M_2, and $i^t \in I$ be an input signal. Moreover, let $nod = \widetilde{\prod}_{m,\rho_{m,di}} \text{None}$ and $allb = \widetilde{\prod}_{n,\rho(n,bi)} \mathsf{T}$. Then, $bint = \mathcal{BO}^m(\omega_2\ (s.m2, int_2))$ and $dint = \mathcal{DO}^m(\omega_1\ (s.m1, allb \tilde{\odot} \mathcal{DI}^n(i)))$ if Assumption 3 holds. In case of Assumption 4, $bint = \mathcal{BO}^m(\omega_2\ (s.m2, \mathcal{BI}^p(i)\ \tilde{\odot}\ nod)$ and $dint = \mathcal{DO}^n(\omega_1\ (s.m1, int_1))$.

A. Replication

The replication operator is the first non-standard composition that we introduce. The goal is the controlled, parallel execution of r copies of a component while maintaining the external input and output interfaces. When we summarise the case studies in Section VI, we present examples how this operation is applied. The basic schematics is depicted in Fig. 1(b).



(a) Sequential Comp. (b) Replication (c) PCI Express Study

Fig. 1. Overviews of Composition Operators and PCI Express Case Study

The four main components are a multiplex function mux, an arbitration function arb, the state machine M to be replicated, and and additional component opt with initial state opt^0 and step function δ_{opt}. An instance of the operator for a state machine M is given by $\mathbf{rep}(r, OPT, mux, arb)$ where

- r is the number of replications,
- $OPT = (Opt, opt^0 \in Opt, \delta_{opt})$ is the optional part,
- $mux : Opt \times \mathcal{DI}^n \to ([1,r] \to \mathcal{DI}^n)$ is the multiplex function, and
- $arb : Opt \times ([1,r] \to \mathcal{DO}^m) \to (\!|\, w : [1,r],\, do : \mathcal{DO}^m \,|\!)$ is the arbitration function.

Definition 10: Let M be the state machine to be replicated. Then, $M_r = \mathbf{rep}(n, OPT, mux, arb)\, M$, is given by:

$$
\begin{aligned}
S_r &= (\!|\, ms : [1,n] \to S,\ opt : Opt \,|\!), \quad I_r = I, \quad O_r = O \\
s0_r &= (\!|\, ms = (\lambda w \in [1,n].s0),\ opt = opt^0 \,|\!) \\
\delta_r &= \lambda s.\, \lambda i.\, \text{let} \\
&\qquad dii = (mux(s.opt, \mathcal{DI}^n(i))) \\
&\qquad doi = \lambda w.\, \omega(s.ms\ w, (\widetilde{\prod}_{m,\rho_{m,bi}} \mathsf{T}) \tilde{\odot} dii\ w) \\
&\qquad arbo = arb\,(s.opt, \lambda w.\, \mathcal{DO}^m(doi\ w)) \\
&\qquad bii = sel_a\,(\mathcal{BI}^m(i), arbo.w) \\
&\qquad ms' = \lambda w.\, \delta(s.ms\ w, (bii\ w) \tilde{\odot} (dii\ w)) \\
&\quad \text{in } (ms = ms',\ opt = \delta_{opt}(s.opt)) \\
\omega_r &= \lambda s.\, \lambda i.\, \text{let} \\
&\qquad dii = (mux(s.opt, \mathcal{DI}^n(i)) \\
&\qquad doi = \lambda w.\, \omega(s.ms\ w, (\widetilde{\prod}_{m,\rho(m,bi)} \mathsf{T}) \tilde{\odot} dii\ w) \\
&\qquad bo = sel_m\,(\lambda w.\, \mathcal{BO}^n(doi\ w), dii.w)) \\
&\quad \text{in } bo \,\tilde{\odot}\, arb\,(s.opt, \lambda w.\, \mathcal{DO}^m(doi\ w))
\end{aligned}
$$

where $sel_m : ([1,n] \to \mathcal{BO}_n \times [1,n]) \to \mathcal{BO}_n$, $sel_a : (\mathcal{BI}_m \times [1,m]) \to ([1,m] \to \mathcal{BI}_m)$ are the busy select functions. Note that the construction requires M to satisfy Assumption 3. We conclude the specification of the replication operator by stating two assumptions which ensures that the construction *makes sense*. The first one states that the multiplex function selects a unique internal component for a given data element.

Assumption 5 (Valid Multiplex Function): Let $id = mux\,(opt, i)$. A valid multiplex function mux satisfies:

$$
\begin{aligned}
\exists!\ w \in [1:r]. \quad & (id\ w = i) \wedge \\
& \forall k \neq w.\, (id\ k = \widetilde{\prod}_{n,\rho(n,di)} \text{None})
\end{aligned}
$$

The second assumption states similarly that the output of the arbitration function is coherent.

Assumption 6 (Coherent Arbitration Function): Let $(w, do) = arb\,(opt, idos)$. A valid arbitration function arb satisfies:

$$\forall idos \in [1,r] \to \mathcal{DO}^m.\, (do = idos\ w)$$

B. Multiplex/Arbitrate

The multiplex/arbitrate composition is a similar, but more general, construct as the replicate operator. The goal is to parallelise n arbitrary components in a structured way. Thus we are not restricted to an instantiation with n copies of a state machine. In order to define a generic construction and allow arbitrary components, the input and output interface have to change. We only allow components with the same number of input and output interfaces. We also assume some *logical* relation between the i-th data component of each state machine. Then, the i-th external input or output interface provides data elements from the union of the all the i-th internal interfaces.

Moreover, the multiplex component does not have to be unique anymore and can select more than one internal state machine. For example, it might split a data element and input the two parts to two internal state machines. Similarly, the arbitration function may select more than one internal component, but is still only allowed to produce a single output, of course. Again, an idea may be the data elements that have been split by the multiplex function are combined again. Because of space restrictions, we omit the formal definition here as it is analogous to Definition 10.

C. Communication Channels

To conclude the section on component composition, we introduce models of communication channels. Having modelled the endpoints, we need to be able to interconnect them. We introduce both: a model for point-to-point topologies, and a model for communication busses.

The goal is to specify interconnects with a transmission delay $d \in \mathbb{N}$ and a capacity of $c \leq d \in \mathbb{N}$ data elements. To define such a point-to-point channel, we can use previously specified components: sequential composition, a buffer, a data modification, and a routing component. The idea is as follows: we use a buffer of size c to provide the capacity. To model the delay, we first use a data modification to add a sequence number $seq \in [0, d)$ to a data element. After the buffer, we use a routing component to check if a counter value $cnt \in [0, d)$ is equal to the sequence number and only in case it is equal, the data is passed on. Both the sequence number and the counter are increased in every time step, whether there is a new data element or not. Note that this works because a buffer generates a delay of at least one and we ensure that at all times the sequence number is equal to the counter value.

Definition 11 (Point-to-Point Channel): Let M_{seq} be the state machine obtained from instantiating a data modification unit with $Opt = [0,d)$, $opt^0 = (\!|\, opt = 0 \,|\!)$, $\delta_{opt} = \lambda s, i.\, s.opt + 1$, and $f = \lambda s, x.\, (x, s.opt)$. Moreover, let $M_{buf} = (\mathcal{D} \times [0,d), c)\, Buf$. Finally, let M_{del} be a routing unit with $n = m = 1$, $Opt = [0,d)$, $opt^0 = (\!|\, opt = 0 \,|\!)$,

and $\delta_{opt} = \lambda s, i.\ s(\!| \ opt := s.opt + 1 \ |\!))$. The routing functions are $f_b = \lambda s, i.\ s.opt \neq \mathbf{snd}(i.di)$ and $f_d = \lambda s, i.$ if $\neg f_b\ (s, i)$ then Some $(\mathbf{fst}\ (i.di))$ else None. Then, $(d, c, \mathcal{D})\ Chan = (M_{seq}\,\mathbf{seq}\,M_{buf}\,\mathbf{seq}\,M_{del})$.
In order to define a communication bus, we use the channel and simply compose it with two more routing units to generate the inputs and outputs to the channel.

Definition 12 (Communication Bus): Let $g : I^v \to I$ be an arbitration function to select among v bus inputs the one that is allowed to use the bus. A bus with delay d, capacity $c \leq d$, inputs v, and outputs w, $(d, c, v, w, g)\ Bus$, is constructed as $M_{arb}\,\mathbf{seq}\,(d, c)\ \ Chan\,\mathbf{seq}\,M_{mux}$. M_{arb} is a routing instance with $(n, m) = (v, 1)$ using g to select an input. M_{mux} is a routing instance with $(n.m) = (1, w)$ that forwards the input to all w outputs.

V. Verification and Automatic Tool Support

In this section, we detail our verification approach and the generic correctness properties of the framework components. By integrating a model checker and using automated theorem prover, we react to concerns regarding the feasibility of the theorem proving approach. The support for automated verification tools aims mainly at simplifying the discharging of the local assumptions. But we were also able to apply them to parts of the generic correctness argumentation.

Here, we focus on the integration and use of the model checker. Since our modelling approach is based on state machines, we can use NuSMV to reason about LTL and CTL properties. The main use of the model checker is to discharge the local assumptions when applying one of the specific composition operators. A good example for the merits of the model checker is the arbitration function in the replication composition. As we will see in Section V-B, we require the arbitration function to be fair with respect to the data signals. Instead of a tedious induction proof using interactive theorem proving, we can use the model checker: in the simplest case we can check whether a given arbitration function satisfies

$$\mathbf{G}(i.di_k = \text{Some } x) \implies \mathbf{F}\ (arb.w = i)$$

where $\mathbf{G}$ and $\mathbf{F}$ are the standard LTL operators for *globally* and *finally*.

A proven LTL or CTL property can easily be translated to the execution trace semantic from Definition 2. This way, we can integrate model checking in the theorem proving workflow. Since we use NuSMV mainly to discharge local assumptions when we apply the framework to a specific protocol, the 'end-user' verification part benefits from the integration. It is a very promising first step to the final goal of reducing the theorem prover to a knowledge management system only and large parts of the framework application steps can already be automated using NuSMV.

A. Basic Components

In order to argue about correctness in a reasonable way, we have to introduce an environment assumption first. We assume in the following that the busy signal provided by the environment, i. e. by the host system, is fair in the sense that it is not constantly active. Thus, the environment allows progress. Assumption 7 formulates this by stating that a busy signal is only constantly active for a finite time interval.

Assumption 7 (Fair busy Signals): For all external busy signals b holds: $\forall t.\ \exists k.\ \neg b^{t+k}$
Note that this is a common assumption for inputs with a semantics similar to the busy signal. We can easily see that the definition of the buffer satisfies Assumptions 1, 2, 3, and 4. Here, we show that the buffer provides stable output signals as long as the busy input is active. It is basically a conclusion from Assumption 1. Then, we state the two main buffer theorems: a liveness and an ordering (FIFO) property.

Lemma 1 (Stable Buffer Outputs): Given a generic buffer $B = (S, I, O, s0, \delta, \omega)$ and an input signal $i^t \in I$, B satisfies:

$$\forall x \in \mathbf{dom}(do, O).\ bi^t \wedge (do^t = \text{Some } x) \implies \exists k.\ \neg bi^{t+k} \wedge (\forall k' \leq k.\ do^{t+k'} = \text{Some } x)$$

Proof: With the integration of the NuSMV model checker, the lemma is automatically shown by re-stating it as an LTL formula $((do = \text{Some } x)\,\mathbf{Until}\,(\neg bi))$. It is then easily translated to our execution trace semantics in HOL. ■

Theorem 1 (Buffer Liveness): Given a fair busy signal, a buffer satisfies the following liveness property:

$$\forall x \in \mathbf{dom}(di, I).\ \neg bo^t \wedge (di^t = \text{Some } x) \implies \exists k.\ (do^{t+k} = \text{Some } x)$$

Proof: This theorem can be automatically shown with NuSMV using the assumption of a fair environment. We show that $\neg bo \wedge (di = \text{Some } x) \implies \mathbf{F}(do = \text{Some } x)$ and translate it to the execution trace semantics. ■
Note that for a simple buffer, $\mathbf{dom}(di, I) = \mathbf{dom}(do, O)$ holds since the basic buffer construct does not implement any data modification. Therefore, we can quantify over $\mathbf{dom}(di, I)$ and argue about the data output.

Theorem 2 (Buffer FIFO Property): A buffer preserves the ordering of its input data. Let $t < t'$ such that $\neg bo^t$ and $\neg bo^{t'}$, then it holds that:

$$\forall x, y \in \mathbf{dom}(di, I).\ (di^t = \text{Some } x) \wedge (di^{t'} = \text{Some } y) \implies \exists k, k'.\ (do^{t+k} = \text{Some } x) \wedge (d_o^{t+k+k'} = \text{Some } y)$$

where the delay values k, k' are given by Theorem 1.

Proof: The liveness part of the statement is shown with Theorem 1. Since buffers are modelled using lists, the ordering property is shown using the ordering property of lists. ■

B. Composition Operators

Given the correctness of the basic building blocks, we need to argue about the composition operators. Our main goal is to show that the properties for the basic components are *preserved* by the compositions. Informally, the idea is that if a component satisfies a correctness property P, we aim at showing that a composed system satisfies a correctness property P' that can be derived from P only using the construction of the composition.

For the parallel composition, the correctness property is straightforward: the composed system satisfies the conjunction of the individual correctness properties. Since parallel composition only executes the two state machines simultaneously without any control or data modification, one can easily see that this is the case.

Lemma 2 (Parallel Composition Correctness): Given state machines M_1, M_2 and corresponding input signals $i_1^t \in I_1$ and $i_2^t \in I_2$. Moreover, let $i = \lambda t \in \mathbb{N}.\ (\!|\, m_1 = i_1^t,\ m_2 = i_2^t \,|\!)$ be the input signal for the parallel composition $M_1 \mathbf{par} M_2$. Then, the following holds:

$$\begin{aligned} &\langle M_1, i_1\rangle \models_{\mathcal{A}_1} P_1 \wedge \langle M_2, i_2\rangle \models_{\mathcal{A}_2} P_2 \\ &\quad \implies \langle M_1 \mathbf{par} M_2, i\rangle \models_{\mathcal{A}_1 \cup \mathcal{A}_2} P_1 \wedge P_2 \end{aligned}$$

Proof: The proof is straightforward by applying the definition of parallel composition (Definition 7). ■

The corresponding lemma for the sequential composition is slightly more complicated since not every input or output of the individual system is still an external input or output in the composed system (cf. Fig. 1(a)). Thus, we need to make the respective substitutions in the correctness statements. To describe a substitution of x by y in a formula P in the following, we use the common notation $P[x/y]$.

Lemma 3 (Sequential Composition Correctness): Given M_1, M_2 and corresponding input signals $i_1^t \in I_1$, $i_2^t \in I_2$. Let *bint* and *dint* be the internal signals from Definition 9, and let $i = \lambda t \in \mathbb{N}.\ \mathcal{BI}^p(i_2^t)\tilde{\odot}\mathcal{DI}^n i_1^t$ be the input signal to the sequential composition $M = M_1 \mathbf{seq} M_2$. Then, it holds:

$$\begin{aligned} &\langle M_1, i_1\rangle \models_{\mathcal{A}_1} P_1 \wedge \langle M_2, i_2\rangle \models_{\mathcal{A}_2} P_2 \\ &\quad \implies \langle M, i\rangle \models_{\mathcal{A}} P_1[\mathcal{BI}(i_1^t)/bint] \wedge P_2[\mathcal{DI}(i_2^t)/dint] \end{aligned}$$

where $\mathcal{A}$ is the union of $\mathcal{A}_1$ and $\mathcal{A}_2$ with the respective input signal substitutions.

Proof: Similar to the proof of Lemma 2, the proof is basically an application of the definition of sequential composition (Definition 8). The proof can be obtained using automatic theorem proving via sledgehammer. ■

Next, we will provide generic correctness results for the replication operator. We will state assumptions that have to be discharged when the composition is instantiated. Given these assumptions, we can show a generic correctness theorem. Since the replication operation is more restrictive than the multiplex/arbitrate composition, we can derive more correctness properties for the former. Therefore, we also give the former preference over the latter in the case studies wherever possible.

The assumptions for the replication operator can be summarized as: (i) the inner components are correct and ensure liveness, (ii) the multiplex function is *correct*, i.e. it multiplexes valid inputs to some inner component (Assumption 5), and (iii) the arbitration is *fair* with respect to an active data signal from an inner component. The following assumption states the first point. We omit the fairness of the arbitration function here due to space limitations.

Assumption 8 (Inner Component): Let M be the state machine to be replicated using the replication operator. Then, M has to satisfy the busy-independent output assumption and has to provide stable output signals, i. e. $\forall i \in I.\ \langle M, i\rangle \models_{\mathcal{A}}$ Assumptions 1 and 3, where $\mathcal{A}$ is the fair environment assumption. Moreover, M has to satisfy liveness:

$$\begin{aligned} &\forall i \in [1, n].\ \forall x \in \mathbf{dom}(di_i, I).\ \neg b_o^t \wedge (di_i^t = \text{Some } x) \\ &\implies (\exists j \in [1, m], k \in \mathbb{N}.\ do_j^{t+k} = \text{Some } f_{i,j}(x)) \end{aligned}$$

where $f_{i,j} : \mathbf{dom}(di_i, I) \rightarrow \mathbf{dom}(do_j, O)$ is a potential data transformation applied by the inner component.

The following theorem states that given Assumption 8 and the assumptions on the multiplex and arbitration functions, the derived system satisfies liveness

Theorem 3 (Correctness of Replication): If the inner state machine satisfies Assumption 8, the system obtained using the replication operator satisfies this assumption again if the multiplex and arbitration functions ensure the previously mentioned assumptions.

Proof: The Isabelle proof of Theorem 3 is mainly obtained by unfolding definitions and assumptions. An induction is needed to conclude the stable input signals for the time interval from Assumption 8 and the fairness of the arbitration function. ■

VI. Case Studies

The development of the framework was driven by the work on two case studies covering rather different protocols: first, the ARM AMBA High-performance Bus (AHB) protocol, an arbiter-based master-slave bus protocol for system on chips. Second, the PCI Express protocol, an off-chip point-to-point high-performance protocol implementing many sophisticated features of current and future on-chip communication protocols. By choosing two case studies covering a wide range of protocol features as well as bus and point-to-point network topologies, we show the utility and breadth of the framework.

A. AMBA High-performance Bus

The AHB protocol is a bus protocol where masters access data stored in slaves, all connected to a bus. Bus access is regulated by an arbiter. The bus itself consists of an address and data bus. Each transfer is split into two, in the simple case, consecutive phases: an address and a data phase. In two steps, we add pipelined transfers and burst support.

Pipelined bus transfers are realised using the replication operator and executing two copies of the sequential master in parallel. The address and data bus outputs of the sequential masters are arbitrated such that address and data phases on the bus are pipelined. Burst transfers are added to either a sequential or pipelined master by the sequential composition of a control flow instance and the master. The idea is to generate a sequence of transfers with an incrementing address counter.

We were able to model two crucial and widely-used features of bus protocols with only two framework components.

B. PCI Express

The PCI Express case study was more extensive than the initial AHB study. Our goal was to investigate the approach using an industrial-sized protocol which implements a series of features that are used in modern multi- and many-core communication architectures. The protocol is specified using three stack layers, each implementing an abstraction layer: a transaction layer (TL), a data-link layer (DLL), and a physical layer. Our case study considered the two upper layers with a focus on the TL. An overview of the case study is shown in Fig. 1(c).

As an example of reducing a complex feature to a small set of framework operations, we outline the receiver part of the flow control system. Using simple buffer, a small instance of the multiplex/arbitrate composition, and finally an instance of the replication operator, we are able to construct a receiver model supporting flow control. It is a very nice example of constructing a complex system incrementally starting with a very simple one in structured way, applying the generic correctness results to verify each modelling step.

With the small set of composition operators and basic building blocks presented here, we were able to model almost all of the features mentioned in Fig. 1(c). Only for the ACKNAK protocol we used an ad-hoc modelling approach. But we were still able to encapsulate the feature in a transformation and use sequential composition to integrate it in the DLL model.

VII. Conclusion

We have formalized a framework for the modelling and verification of on-chip communication protocols in the Isabelle/HOL theorem prover. Our modelling approach is based on abstract state machines and we specify initial, basic building blocks. Using composition rules, especially the replication and multiplex/arbitrate operators, we can incrementally compose more complex systems. In previous contributions, we have shown how to apply these principles to model protocol features independently. With a small set of basic building blocks and composition rules, we were able to model a broad variety of protocol features. The framework is flexible enough so that it is applicable to two very different protocol types covered by the case studies.

We managed to reduce the verification effort by spreading it over the modelling process and integrating automated tools into the methodology such as the NuSMV model checker. We also prove generic correctness properties for the composition rules so that we can restrict the verification to discharging local assumptions when we use the framework to model concrete feature extensions. By reducing the manual theorem proving parts compared to out previous case studies, we tackled frequent concerns regarding the usability of our approach. Future work includes an even further reduction of the theorem proving, ideally to a point where automated tools are sufficient to apply the framework to a specific case study.

Future work in the short-term focuses around two points: integrating even more automatic tools and linking the models to hardware descriptions. For the former, we plan on investigating the integration of an SMT Solver in the approach, for example the current Isabelle version provides a link to the Z3 SMT solver. Also the integration of a SAT solver will be considered. To link actual hardware descriptions to the models, we aim at further integrating the IHaVeIt interface in Isabelle 2009 as it also provides generators for Verilog and VHDL code.

Our larger-scale, long-term aim is to provide a feasible approach to modelling and verification of complex on-chip protocols as an alternative to monolithic, ad-hoc modelling and post-hoc verification. We aim at increasing the efficiency of the model building process, and providing a final model that has significant merits against ad-hoc models. The merits of our models is that they are already functionally verified and independent from the actual implementation or design architecture which can act as a longer-term reference model.

VIII. Acknowledgements

This work is funded by the Engineering and Physical Sciences Research Council and a donation from Intel Corporation.

References

[1] P. Böhm, "Incremental Modelling and Verification of the PCI Express Transaction Layer," in *MEMOCODE'09*. IEEE, 2009, pp. 36–45.

[2] P. Böhm and T. Melham, "A Refinement Approach to Design and Verification of On-Chip Communication Protocols," in *FMCAD'08*. IEEE, 2008, pp. 136–143.

[3] *AMBA Specification Revision 2.0*, ARM, 1999.

[4] PCI-SIG, *PCI Express Base Specification Revision 2.0*, December 2006.

[5] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS vol. 2283, Springer, 2002.

[6] S. Tverdyshev, "Formal verification of gate-level computer systems," Ph.D. dissertation, Saarland University, Computer Science Department, 2009. [Online]. Available: http://www-wjp.cs.uni-saarland.de/publikationen/Tv09.pdf

[7] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.

[8] T. Nipkow and K. Slind, "I/O automata in Isabelle/HOL," in *TYPES'94*, ser. LNCS, vol. 996. Springer, 1995, pp. 101–119.

[9] L. Helmink, P. A. Sellink, M., and W. Vaandrager, F., "Proof-checking a data link protocol," in *TYPES'93*. Springer, 1994, pp. 127–165.

[10] V. Luchangco, E. Söylemez, S. J. Garland, and N. A. Lynch, "Verifying timing properties of concurrent algorithms," in *Formal Description Techniques VII*. Chapman & Hall, Ltd., 1995, pp. 259–273.

[11] S. M. Suhaib, D. A. Mathaikutty, S. K. Shukla, and D. Berner, "Xfm: An incremental methodology for developing formal models," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 4, pp. 589–609, 2005.

[12] J. R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen, "The B-method," in *VDM'91*, ser. LNCS, vol. 552. Springer, 1991.

[13] J.-R. Abrial, D. Cansell, and D. Méry, "A Mechanically Proved and Incremental Development of IEEE 1394 Tree Identify Protocol," *Formal Aspects of Computing*, vol. 14, no. 3, pp. 215–227, April 2003.

[14] J. Schmaltz and D. Borrione, "A functional formalization of on chip communications," *Form. Asp. Comp.*, vol. 20, no. 3, pp. 241–258, 2008.

[15] X. Chen, S. M. German, and G. Gopalakrishnan, "Transaction Based Modeling and Verification of Hardware Protocols," in *FMCAD'07*. IEEE, 2007, pp. 53–61.

[16] F. Müffke, "A Better Way to Design Communication Protocols," Ph.D. dissertation, University of Bristol, May 2004. [Online]. Available: http://www.cs.bris.ac.uk/Publications/Papers/2000199.pdf

[17] S. Finn and M. Fourman, "The LAMBDA Logic. Abstract Hardware Limited, September 1993. In LAMBDA 4.3 Reference Manuals." 1993.

[18] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991.

[19] K. L. McMillan, "A compositional rule for hardware design refinement," in *CAV '97*. Springer, 1997, pp. 24–35.

[20] J. Schmaltz, "A formal model of clock domain crossing and automated verification of time-triggered hardware," in *FMCAD '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 223–230.

[21] E. Alkassar, P. Böhm, and S. Knapp, "Correctness of a fault-tolerant real-time scheduler and its hardware implementation," in *MEMOCODE'08*. IEEE Computer Society, June 2008, pp. 175–186.

[22] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray, "A survey of hybrid techniques for functional verification," *IEEE Des. Test*, vol. 24, no. 2, pp. 112–122, 2007.