A Refinement Approach to Design and Verification of On-Chip Communication Protocols

Peter Böhm, Tom Melham
Oxford University Computing Laboratory
Oxford, OX1 3QD, England
Email: {peter.boehm,tom.melham}@comlab.ox.ac.uk

***Abstract*—Modern computer systems rely more and more on on-chip communication protocols to exchange data. To meet performance requirements these protocols have become highly complex, which usually makes their formal verification infeasible with reasonable time and effort.**

We present a new refinement approach to on-chip communication protocols that combines design and verification together, interleaving them hand-in-hand. Our modeling framework consists of *design steps* and *design transformations* formalized as finite state machines. Given a verified design step, transformations are used to extend the system with advanced features. A design transformation ensures that the extended design is correct if the previous system is correct.

This approach is illustrated by an arbiter-based master-slave communication system inspired by the AMBA High-performance Bus architecture. Starting with a sequential protocol design, it is extended with pipelining and burst transfers. Transformations are generated from design constraints providing a basis for correctness-by-design of the derived system.

I. Introduction

Modern computer systems rely more and more on highly complex on-chip communication protocols to exchange data. The enormous complexity of these protocols results from meeting high-performance requirements. Communication can be pipelined, data may be distributed into burst parts, and burst transfers may be split. Protocol control can be distributed and there may be non-atomicity or speculation. Moreover, some components may have separate clocks or adjustable clock frequencies, requiring asynchronous communications. These complexities arise in many important domains, such as multicore architectures, system-on-chip, or network-on-chip designs. Although the effort of chip manufacturers to validate or even formally verify their designs has increased, the complexity of communication protocols usually makes their formal verification infeasible with reasonable time and effort.

In this paper, we present a new approach to the design and formal verification of on-chip communication protocols based on refinement steps. The approach can be summarized as follows. The modeling framework is based on two central concepts, *design steps* and *design transformations*. We start with a core design step for a basic protocol that can be formally verified with reasonable effort. This is then extended with advanced features step-by-step to meet performance demands, handle asynchronous communication, or improve fault-tolerance. These extensions are realized by mathematical transformations of a previous design step, rather than constructing a new design. The correctness of an extended design is obtained from the correctness of the previous version and either the transformation itself (*correctness-by-design*) or a refinement or simulation relation between the two versions.

Verification by stepwise refinement is in general, of course, not new and there is a rich literature going at least back to Dijkstra [1], [2] and Wirth [3]. Our contributions are the application of this methodology to protocol verification, the design of a generic modeling framework specifically for this application, and the development of technical details of the specific optimising transformations we are investigating.

As a case study, we illustrate this refinement approach with an arbiter-based master-slave communication system inspired by the AMBA High-performance Bus architecture (AHB) [4]. The initial design step is a basic, sequential communication system. This system is extended with pipelined transfers and burst support. The corresponding transformations are partly derived automatically based on a behavioural constraint specification. This provides a basis for a correctness-by-design argumentation which is used within the demonstration of the correctness of the extended systems.

A. Related Work

Hardware verification based on refinement checking or simulation relations has a long history. Abadi and Lamport [5] show the existence of refinement mappings in their widely-cited article. McMillan [6] proposes a compositional rule for hardware verification based on local refinements which can be efficiently model checked. Aagaard *et al.* [7] present a framework for microprocessor correctness statements based on simulation relations.

Dill *et al.* [8] propose the protocol description language and verifier Murφ and its application to a industrial cache coherency as well as a link-level protocol. Eiríksson [9] and more recently German [10] emphasize the need for integration of formal verification into the (industrial) design process.

Most existing work on communication protocol verification addresses the correctness of specific protocols. For example, Roychoudhury *et al.* [11] present a formal specification of the AMBA protocol. They use an academic protocol version and verify design invariants using the SMV model checker. Amjad [12] verifies latency, arbitration, coherence, and deadlock freedom properties of a simplified AMBA model. Schmaltz *et al.* [13] present initial work on a generic network on chip

978-1-4244-2735-2/08/$25.00 ©2008 IEEE

model as a framework for correct on-chip communication. They identify key constraints on architectures and show protocol correctness provided they are satisfied.

All these approaches rely on a post-hoc protocol verification, which is a key difference from the methodology presented here. Even the framework in [13] relies on a post-hoc verification of protocol properties against their constraints.

Chen *et al.* [14] propose a modular, refinement based approach to verify transaction based hardware implementations against their specification models. They use a cache coherency protocol to illustrate their methodology. Müffke [15] 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. Claiming that the generated interfaces are correct by construction in terms of their specification he neither addresses the protocol correctness itself nor the verification of the implementation against the specification.

The basic idea of our approach is similar to Intel's integrated design and verification (IDV) system [16]. The IDV system justifies its transformations by a *local proof* using simple equivalence checking. We expect, however, that transformations tailored for high-performance on-chip communication protocols will need more intricate refinement steps than can be handled by equivalence checking.

II. Modeling Framework

In this section we present a generic modeling framework for the design and verification of arbiter-based, master-slave communication protocols. The two basic concepts are *design steps* and *design transformations*. The design process starts with an initial design step that is transformed into subsequent design steps with extended features or functionality.

A. Design Step

A design step is used to represent a verified protocol version during the design process. It has three parts: a specification of the communication system, a definition of a transaction model, and a predicate defining message transfer correctness.

1) Communication System: All communication systems under consideration have three kinds of basic components: a finite numbers of *masters* and *slaves* plus a single *arbiter*. Masters are attached to a host system and they have to react to host requests for data transfers that use the communication system. Slaves have an attached memory system and they respond to requests from masters by accessing it. The arbiter is responsible for granting bus access to a master. They are all specified as finite state machines. Additionally, there is a stateless communication bus between the masters and slaves.

We have a collection of N_M identical masters where $M_i = (I_M, O_M, Q_M, q_M^0, \delta_M, \omega_M)$ for $1 \leq i \leq N_M$. Using conventional notation for finite state machines, I_M and O_M denote the sets of inputs and outputs, Q_M the finite set of states, and q_M^0 the initial state. The state-transition function is given by $\delta_M : (Q_M \times I_M) \rightarrow Q_M$ and the output function is denoted by $\omega_M : (Q_M \times I_M) \rightarrow O_M$. Analogously, we have a collection of N_S identical slaves where $S_i = (I_S, O_S, Q_S, q_S^0, \delta_S, \omega_S)$ for $1 \leq i \leq N_S$ and an arbiter A where $A = (I_A, O_A, Q_A, q_A^0, \delta_A, \omega_A)$.

The communication system is then specified by the parallel execution of those finite state machines, thus again a finite state machine. As we have to refer to the communication bus frequently, we also add the bus as a separate component.

Definition 1 (Communication System CS) *A communication system CS is given by the finite state machine*

$$CS = (I, O, Q, q^0, \delta, \omega, cbus)$$

where I and O denote the sets of external inputs and outputs, $Q = ({Q_M}^M \times {Q_S}^S \times A)$ denotes the state space and q^0 is the initial state. The state-transition function is given by $\delta : (Q \times I) \rightarrow Q$ and $\omega : (Q \times I) \rightarrow O$ is the output function. The bus $cbus$ is given by a collection of signals obtained by combinatorial logic from the outputs of masters and slaves.

Signals are modeled as functions from discrete time to bit vectors of length n, i.e. $\mathbb{N} \rightarrow \mathbb{B}^n$, and sig^t denotes the value of a signal sig at time t.

The transition function of the system applies the transition functions of the components simultaneously in every step.

2) Transaction Model: In order to talk about transfers, we need a concept of a *transaction* which represents a unit of communication between a master and the memory modeled by the slaves. Our concept of a transaction is called *abstract transfer*. It is defined as a function $tr : \mathbb{N} \rightarrow T$ where T is the state space of a single transaction. We refer to the i-th transfer by $tr(i)$. This model encapsulates timing information as the definition of start and end times of a transfer. It is specific to a concrete instantiation and we present examples later.

3) Correctness Predicate: The third part of a design step is the correctness predicate $corr$. In general, this has to relate the execution behaviour of the communication system to a memory model. As every slave has a memory attached, the collection of all slaves provides a global memory spread over the communication system. A global memory model gm is a function over a cycle-accurate time domain $\mathbb{N}$ and an address space Ad to a data element of some data space D, i.e. $gm : (\mathbb{N} \times Ad) \rightarrow D$. The operations allowed can be categorized into *reads* and *writes*.

Given such a model, the correctness predicate has to enforce correctness properties for read transfers (RD), i.e. a transfer initiated by a host of some master requesting a read operation on the communication system, and write transfers (WR), i.e. a transfer initiated by a host requesting a write operation. Additionally, an optional predicate (OPT) may formulate specific protocol properties not 'visible' in standard, single transfers such as pipelining. Thus, the correctness predicate $corr(CS, tr) \subset (Q \times T)$ has the following structure:

$$corr(CS, tr) = RD \wedge WR \wedge OPT$$

Finally, we can summarize a design step as a triple consisting of the afore-mentioned parts:

Definition 2 (Design Step DS) *A design step is given by a triple $(CS, tr, corr)$. We call a design step* valid *iff $corr(CS, tr)$ holds.*

Note that the definition of a valid design step allows meaningless constructions as $corr(CS, tr) = True$. We assume that the correctness predicate argues about the correct message transmission according to the global memory model in a meaningful way. We will see examples of meaningful correctness predicates in Sections III and IV.

B. Design Transformation

A transformation is an operator on a design step to move from one valid design to another. It has three parts: a function θ modifying a given communication system CS, a corresponding abstract transfer model tr_θ, possibly relying on the 'input' transfer model, and a new correctness predicate $corr_\theta$.

Definition 3 (Design Transformation $trans$) *A design step transformation $trans$ is given by the triple $(\theta, tr_\theta, corr_\theta)$.*

The validity of the new correctness predicate $corr_\theta$ is obtained from the transformation function θ and the correctness of the previous design $corr$.

$$\theta \wedge corr(CS, tr) \implies corr_\theta(\theta(CS), tr_\theta)$$

As it stands, the state machine framework just described is rather obvious and unspecific. Our ultimate goal, however, is to elaborate this clear-cut starting point with extra formal structure that captures the generic capabilities and constraints of a broad, but specific, target class of high-performance communications architecture. The case study presented in the remainder of the paper is an example of the kind of experimental investigation that will inform the derivation of this framework.

III. Core Design Step

In this section, we use the notion of a design step to specify the initial protocol design of our case study. The system is a protocol design inspired by the AHB supporting sequential transfers with possible slave-side wait-states for memory operations. It is the initial design step, though its correctness has to be proven initially. As it supports sequential transfers, we prefix the components with *s*, thus $SDS = (SCS, str, scorr)$. This design step is extended with pipelining in Section IV-A and support for burst transfers in Section IV-B.

A. Communication System and Abstract Tranfer

To specify the communication system SCS, we define the the finite state machines of the three kinds of components, and the communication bus. We start with the latter.

Definition 4 (Sequential Communication Bus) *The value of the master-slave communication bus at time t of the sequential system is defined as the following tuple:*

$$(rdy^t, trans^t, wr^t, addr^t, wdata^t, rdata^t) \in \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}^{ad} \times \mathbb{B}^d \times \mathbb{B}^d$$

where the components are:

- *rdy is the bus ready signal.*
- *$trans$ is the signal indicating either an idle transfer (0) or a data transfer (1).*
- *wr says if a data transfer is a read (0) or a write (1) transfer.*
- *$addr$ denotes the address bus of width ad. The address consists of a local address part (the lower sl bits) specifying the memory address within a slave and a device address (the upper $ad - sl$ bits) specifying the currently addressed slave.*
- *$wdata$ denotes the write data bus of width d.*
- *$rdata$ denotes the read data bus of width d.*

We refer to the last two components as the data bus *and to the second to fourth components as the* control bus*.*

Before specifying the remaining components, we first define an abstract sequential transfer. A transfer is split into two main parts: first an address phase $aphase$ and then a succeeding data phase $dphase$. During the former, a master puts the control data on the control bus and the addressed slave has to sample the data at the end of that phase. During the latter, the addressed slave performs its memory operation and either the master provides the data to be written or the slave delivers the read data at the end of that phase. The end of each phase is indicated by an active bus ready signal, i.e. $rdy = 1$.

Definition 5 (Abstract Sequential Transfer) *The i-th abstract sequential transfer $str(i)$ is defined in terms of a grant value $gnt \in [1 : N_M]$, a single bit $isdata \in \mathbb{B}$ indicating a idle or data transfer, and three cycle-accurate time points. The first time point $tg \in \mathbb{N}$ is the time when the bus is granted to the master gnt. The second time point $ta \in \mathbb{N}$ is the time when the address phase ends. The third time point $td \in \mathbb{N}$ denotes the time when the data phase of transfer i ends.*

$$str(i) = (gnt, isdata, tg, ta, td) \in [1 : N_M] \times \mathbb{B} \times \mathbb{N}^3$$

The components are defined as

$$\begin{aligned} gnt &= arb.grant^{tg} \\ isdata &= \begin{cases} 0 & : \textit{idle transfer} \\ 1 & : \textit{otherwise} \end{cases} \\ tg &= \begin{cases} 0 & : i = 0 \\ str(i-1).td & : \textit{otherwise} \end{cases} \\ ta &= \min\{t > tg \mid rdy^t\} \\ td &= \min\{t > ta \mid rdy^t\} \end{aligned}$$

where $arb.grant$ denotes the arbiter configuration component specifying the currently granted master.

From the abstract transfer definition, we can summarize three key protocol characteristics for this core design step: (i) every transfer consists of an address and a data phase, (ii) the end

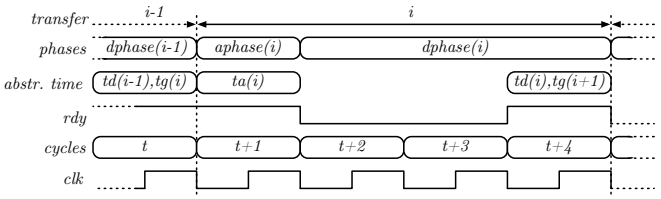

Fig. 1. Sequence of Sequential Transfers

of each phase is defined by the bus signal rdy, and (iii) the bus is granted to some master at every time instant. These characteristics and the definition of an abstract sequential transfer are illustrated in Fig. 1. To ease notation, we define a function $i(u,t)$. It denotes the next transfer such that tg is greater or equal to t and the bus is granted to master u.

$$i(u,t) = \min\{j \mid t \leq str(j).tg \wedge u = str(j).gnt\}$$

In case no such minimum is defined, we say $i(u,t) = -\infty$.

Next we define the transition function $s\delta$. We will specify the state-transition function by the functions for the components, i.e. $s\delta_M$, $s\delta_S$, and $s\delta_A$. In the following we assume that $N_M = 2^k$ for some $k \in \mathbb{N}$ and $N_S = 2^{ad-sl}$.

The arbiter grants the bus to master M_i with $1 \leq i \leq N_M$ by activating $grant[i-1]$ of its output $grant \in \mathbb{B}^{N_M}$. In case no master requests the bus, the arbiter grants the bus to some fixed default-master. In the scope of this project, we abstract the arbiter to a combinatorial circuit relying on an abstract function af generating a new grant vector given the current one and the current request vector. The arbiter updates the $grant$ bit vector at the end of an address phase, i.e. at ta. To store the information whether an active rdy signal represents the end of an address phase or the end of a data phase, we use a flag $aphase$.

If the $grant$ vector is updated at the end of the address phase, the old vector is still required during the data phase to select the correct $wdata$ output from a transmitting master. It is stored as a delayed grant value in $dgrant$. The state of the sequential arbiter at a given time t is then:

$$(grant^t, dgrant^t, req^t, aphase^t)$$
$$\in [1:N_M] \times [1:N_M] \times \mathbb{B}^M \times \mathbb{B} = SQ_A$$

The state-transition function $s\delta_A$ and the output function $s\omega_A$ are obtained directly from the description above.

The slave is similarly straightforward. Its task is to perform read or write accesses to an attached memory system mem. The slave obtains as inputs all bus components except for $rdata$. The $addr$ signal is reduced to the local address $saddr = addr[sl-1:0] \in \mathbb{B}^{sl}$. The upper part of the address bus ($addr[ad-1:sl]$) is used to generate a select input signal sel by an address decoder.

In case a slave is currently addressed for a data transfer, indicated by an active sel and $trans$ input at the end of the address phase, it has to sample the control bus data. Afterwards, the actual memory access is performed during the data phase. During that access the memory system can activate a memory busy signal $mem.busy$. The memory delivers the requested data when $mem.busy$ is low for the first time after the start of the request. We assume that the memory system is busy for only a finite number of cycles. At the end of the memory request, the slave activates the rdy output. It also provides the read data on the $rdata$ output for a read access.

The sequential slave has to generate the rdy signal indicating the end of the address phase, ta, in addition to the rdy signal indicating td. As the address data can be sampled during one cycle, a unit delay register rdy' is used to delay an active rdy single by one cycle. Then, if rdy' and sel are active, the rdy_{out} output is enabled to generate the rdy signal of the bus at ta. Moreover, in case of an $idle$ transmission, the slave just produces an other rdy_{out} signal in the next cycle (specifying td). The configuration of a sequential slave SS_v for $1 \leq v \leq N_s$ at a given time t is defined as the tuple:

$$(state^t, wr^t, addr^t, wdata^t, mem^t)$$
$$\in \{idle, req\} \times \mathbb{B} \times \mathbb{B}^{sl} \times \mathbb{B}^d \times (\mathbb{B}^{sl} \to \mathbb{B}^d) = SQ_S$$

where mem denotes the local memory. Similarly to the arbiter, the slave is realized according to the above description.

The master provides the interface between the communication system and an attached host system. It handles host requests to transfer data. Thus the master has inputs from the host denoted $startreq \in \mathbb{B}$, indicating a transfer request, and host data signals denoted $hwr \in \mathbb{B}$, $haddr \in \mathbb{B}^{ad}$, and $hwdata \in \mathbb{B}^d$ for the respective transfer data. In case the master is not granted the bus, it has to perform a bus request to the arbiter. Additionally, in case there is no data to transmit but the master is granted the bus, it has to initiate an idle transfer in order to meet the protocol requirements. The inputs to master M_u are the $grant[i-1] \in \mathbb{B}$ signal from the arbiter and the signals $rdy \in \mathbb{B}$, $rdata \in \mathbb{B}^d$ from the bus.

As outputs the master provides the signals $trans \in \mathbb{B}$, $wr \in \mathbb{B}$, $addr \in \mathbb{B}^{ad}$, and $wdata \in \mathbb{B}^d$ used to generate the corresponding bus signals. It provides a request signal $req \in \mathbb{B}$ to the arbiter and a busy signal $busy \in \mathbb{B}$ as well as a signal $hrdata \in \mathbb{B}^d$ to the host. The purpose of the $busy$ signal is the following: the correct transmission of a host request is shown if the master is not busy while the transfer is initiated ($startreq^t \implies \neg busy^t$). We call such a host request *valid* and define a predicate $validreq^t = startreq^t \wedge \neg busy^t$.

The configuration of the master consists of a $state$ component, a flag $vreq$ indicating a pending request, and a set of sampling registers. Thus the state of a sequential master SM_u for $1 \leq u \leq N_m$ at a given time t is defined as the tuple:

$$(state^t, vreq^t, lwr^t, laddr^t, lwdata^t, lrdata^t)$$
$$\in \{idle, aph, dph\} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}^{ad} \times \mathbb{B}^d \times B^d = SQ_M$$

where the components are:

- $state$ denotes the automaton state: $idle$ is the idle state, aph the state denoting the address phase, and dph the state denoting the data phase, respectively.
- $vreq$ denotes that a valid request is currently processed.
- $lwr, laddr, lwdata$ denote the local sampling register for the corresponding host data inputs.

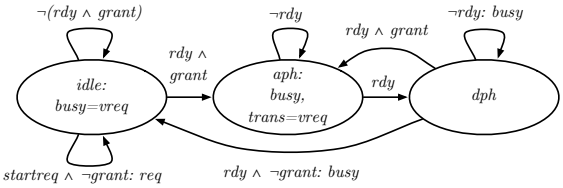

Fig. 2. Sequential Master Control Automaton

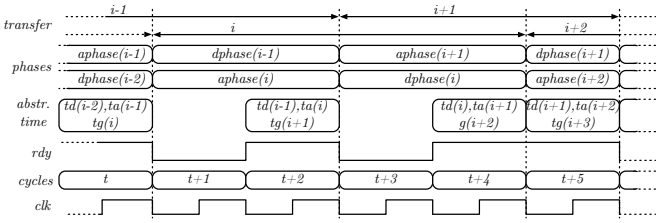

Fig. 3. Sequence of Pipelined Transfers

- $lrdata$ denotes the register used to sample the $rdata$ bus.

The control automaton is shown in Fig. 2 illustrating the update of the $state$ component and the $trans$ output generation. The bus outputs different from $trans$ are simply the respective local components. The local sampling registers are updated in a straightforward way, i.e. $lrdata$ is updated at the end of the data phase in case of a read transfer and the others are updated on a valid host request. The valid request flag $vreq$ is updated according to the following specification:

$$vreq^{t+1} = \begin{cases} 1 & : startreq^t \wedge (idle^t \wedge \neg busy^t) \\ & \quad \vee (data^t \wedge\ grant^t \wedge rdy^t) \\ 0 & : data^t \wedge rdy^t \\ & \quad \wedge \neg(startreq^t \wedge grant^t) \\ vreq^t & : \text{otherwise} \end{cases}$$

This concludes the required parts for the state-transition function $s\delta_M$ and the output function $s\omega_m$.

B. Correctness Predicate

The core design step should realize a simple single-port, read-write memory model. The correctness property $scorr$ relates this memory model to the communication system presented in the previous section. This property says that the communication system behaves like a memory model gm from the view of a host system but with respect to the time points given by tr. Let

$$gm^t(x[ad-1:0]) = SS_{x[ad-1:sl]}.mem^t(x[sl-1:0])$$

and let $1 \leq u \leq N_M$ denote some master index. Then $scorr(SCS, str)$ is defined by:

$$\begin{aligned} validreq_u^t &\implies i(u,t) \neq -\infty \wedge str(i(u,t)).isdata \wedge \\ \neg hwr_u^t &\implies hrdata_u^{td+1} = gm^{tg}(haddr_u^t) \quad \wedge \\ hwr_u^t &\implies gm^{td}(x) = \begin{cases} hwdata_u^t & : x = haddr_u^t \\ gm^{tg}(x) & : \text{otherwise} \end{cases} \end{aligned}$$

Theorem 1 (Valid Core Design Step) *The initial design step SDS is valid, i.e. $scorr(SCS, str)$ holds.*

The proof of this main theorem is based on a series of local correctness properties and communication system invariants. It is detailed in [17].

IV. Transformations

In this section we present transformations to extend the initial design step with pipelining and support for burst transfers.

A. Pipelining

In order to obtain a design with pipelined communication, we devise a design transformation based on duplication of masters. The idea is to execute the address phase of transfer i in parallel with the data phase of the previous transfer $i-1$. This is possible due to the separated control and data buses.

Recall that a transformation consists of a transformation function, a new abstract transfer definition, and a new correctness predicate. We write $ptrans = (pipe, ptr, pcorr)$ for the pipelining transformation.

As we do not introduce new bus signals, the communication bus has the same signals as for the core system.

The basic idea of the abstract transfer model is analogous to the sequential abstract transfer but we have to represent parallel phases. This is achieved by modifying the definition of the bus grant time tg. Note that this is the only change required to that model. Fig. 3 shows a sequence of transfers.

Definition 6 (Abstract Pipelined Transfer) *The i-th abstract pipelined transfer $ptr(i)$ is given by the tuple $(gnt, isdata, tg, ta, td)$ where the components are defined as:*

$$tg = \begin{cases} 0 & : i = 0 \\ ptr.ta & : otherwise \end{cases}$$
$$x = tr_{seq}(i).x \quad for\ x \in \{gnt, isdata, ta, td\}$$

The arbiter is obtained by ignoring the $aphase$ flag of the sequential arbiter and updating the $grant$ vector each time the rdy signal is active. Since the $grant$ vector is updated every time rdy is active, we have to introduce a third grant component denoted $ddgrant$ which is updated with the data from $dgrant$ in exactly the same way as $dgrant$ with $grant$.

The slave is obtained by removing the unit delay rdy'. The pipelined slave only generates a rdy signal at td.

To conclude the definition of the $pipe$ function, we need to specify the transformation for the master. Our goal is to obtain a master supporting pipelined transfers from the master only supporting sequential transfers. The presented transformation is based on the idea of using two sequential masters to construct a single master (see Fig. 4). Since pipelining denotes a process of executing tasks in parallel, using duplicated masters internally is straightforward. But we have to restrict the behaviour of the parallel system by excluding some bad executions, e.g. where both sequential masters are in the address phase, as this obviously leads to a conflict.

The required behavioural constraints are obtained by modifying the inputs to the sequential masters. This is realized by

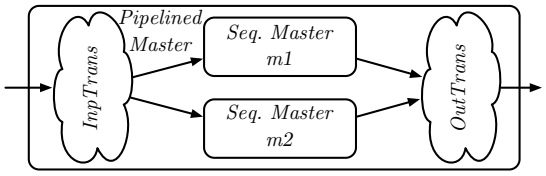

Fig. 4. Basic Concept of Pipelining Transformation

a logic denoted $ITrans$ splitting the inputs into two sets of inputs. Additionally, we need to combine the outputs of the two internal masters to generate the outputs of a single master. This combinatorial function is called $OTrans$.

In the following, we denote the current configuration of the two sequential masters with $sm1 \in SQ_M$ and $sm2 \in SQ_M$. Thus $pm = (sm1, sm2) \in PQ_M$. If we execute two sequential masters in parallel, the state space of the control automaton $pm.state$ is equal to the Cartesian product of the state components of the two sequential masters, i.e. $pm.state \in (sm.state \times sm.state)$.

The key question is which behaviours have to be excluded from the purely interleaved execution of both control automata. We define the desired behaviour in term of constraints regarding the control automaton. We have three kinds of constraints: *state*, *transition*, and *signal* constraints. The first specifies static state constraints, i.e. not reachable states, whereas transition constraints restrict the active, outgoing transitions of a current state. Signal constraints restrict the output signals of the control automaton. We aim at the following constraints:

- State constraints: (i) no *phase contention*, i.e.

$$pm.state \notin \{(aph, aph), (dph, dph)\}$$

and (ii) $sm2$ is only used for pipelining, i.e.

$$pm.state \in \{(idle, aph), (aph, aph), (dph, aph)\} \implies pm.state = (dph, aph)$$

- Transition constraints: *valid steps* of the automaton, i.e.

$$(s,t) \rightsquigarrow (s',t') \wedge s \neq s' \implies t \neq t' \vee t = idle$$
$$(s,t) \rightsquigarrow (s',t') \wedge t \neq t' \implies s \neq s' \vee s = idle$$

- Signal contraints: $sm2$ never requests the bus, i.e. $\neg req_2$

In order to satisfy these constraints, we analyse the product automaton of the two sequential masters. The control automaton of a single master is depicted in Fig. 2. We aim at deriving the constraints to the inputs in an algorithmic way. Using a graph-based algorithm, specifying edges which are not valid according to the constraints above, we obtain predicates to restrict the inputs of the second sequential master.

Let $pmi = (rdy, grant, rdata, startreq, hinp)$ denote the inputs of a master with $hinp = (hwr, haddr, hwdata)$. Then we define the function $ITrans$ as the tuple $(pmi, (rdy, grant_2, rdata, startreq_2, hinp))$ where

$$grant_2 = grant \wedge (sm1.state = aph)$$
$$startreq_2 = startreq \wedge grant_2$$

The outputs are obtained in a straightforward way. The sequential master which is currently in the aph or dph state provides the corresponding bus outputs. Additionally, the req signal to the arbiter is only triggered by $sm1$.

The only non-obvious computation is the $busy$ signal. Obviously, the pipelined master has to be busy if both sequential masters are busy. Additionally, the second sequential master is not allowed to request the bus. Therefore we have to enable the busy signal in cases where the first master is busy and the second master would have to request the bus after a $startreq$ signal, i.e. if the bus is not granted anymore.

Finally, we obtain the transformation for the step function of the master. Let $pm = (sm1, sm2) \in PQ_M$ and $pmi \in PI_M$ denote the inputs to the master. Given that $(smi1, smi2) = ITrans(pmi)$, the transformation $pipe_M$ is defined by:

$$pipe_M(s\delta_M)\ (pm, pmi) = (s\delta_M(sm1, smi1), s\delta_M(sm2, smi2))$$

To complete the specification of the pipeline transformation, we have to define the correctness predicate $pcorr$ and argue about its validity. The correctness predicate of the pipelined system is basically the same as for the core system but we add an optional property stating that the transfer is pipelined. Let gm be defined as in $scorr$ (see Section III-B).

$$validreq_u^t \implies i(u,t) \neq -\infty \wedge ptr.isdata \wedge$$
$$\neg hwr_u^t \implies rdata_u^{td+1} = gm^{tg}(haddr_u^t) \quad \wedge$$
$$hwr_u^t \implies gm^{td}(x) = \begin{cases} hwdata_u^t & : x = haddr_u^t \\ gm^{tg}(x) & : \text{otherwise} \end{cases} \quad \wedge$$
$$[ta : td] = [ptr(i(u,t)+1).tg : ptr(i(u,t)+1)).ta]$$

The main part of its correctness proof is based on a correctness-by-design argumentation of the pipelined master.

B. Burst Support

Next we specify a transformation providing burst transfers applicable to either the core or the pipelined design step. It is general enough that there are only few cases where one has to distinguish whether a sequential or a pipelined design is extended. Again, we will start with the specification of the transformation function and the abstract transfer model. We write $btrans = (bst, btr, bcorr)$ for this transformation.

A burst transfer of size $bsize$ is a transfer starting at address $addr$ transferring all data address $addr$ to $addr + bsize - 1$. We support burst transfers of arbitrary but fixed length up to a maximum of $2^b - 1$ such that $bsize \in \mathbb{B}^b$. The length of a specific transfer is specified during the host request.

In the following, we denote the configuration of a burst master BM_u at time t by bm_u^t, of a sequential master SM_u by sm_u^t, and of a pipelined master PM_u by pm_u^t, respectively.

Next, we define the new abstract transfer model. The changes are more complex than the changes from sequential to pipelined transfers. The basic idea is to add a component bst to indicate a burst transfer together with a field $bsize$ specifying the size. The end of the address phase is now not only a single time point but a partial function assigning an address phase end time to every *sub-transfer* $n \in [0 : bsize(i) - 1]$.

Definition 7 (Abstract Burst Transfer) *The i-th abstract burst transfer $btr(i)$ is defined as the tuple*

$$(gnt, isdata, tg, ta, td, bst, bsize) \\ \in [1 : N_M] \times \mathbb{B} \times \mathbb{N} \times (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \times \mathbb{B} \times \mathbb{B}^b$$

where the components are:

$$gnt = \{s,p\}tr(i).gnt$$
$$isdata = \{s,p\}tr(i).isdata$$
$$tg = \begin{cases} 0 & : i = 0 \\ btr(i-1).ta(btr(i-1).bsize) & : i > 0 \wedge pipe \\ btr(i-1).td & : i > 0 \wedge seq \end{cases}$$
$$ta(n) = \begin{cases} \min\{t > tg \mid rdy^t\} & : \neg bst \vee n = 0 \\ \min\{t > ta(n-1) \mid rdy^t\} & : bst \wedge \\ & \quad 1 \leq n \leq bsize - 1 \\ \textit{undefined} & : otherwise \end{cases}$$
$$td = \begin{cases} \min\{t > ta(0) \mid rdy^t\} & : \neg bst \\ \min\{t > ta(bsize-1) \mid rdy^t\} & : bst \end{cases}$$
$$bst = bm_{gnt}^{ta(0)}.bst$$
$$bsize = bm_{gnt}^{ta(0)}.bsize$$

The main difference between this definition and the previous one is the case split on actual burst transfers. For a non-burst transfer the above definition resolves to one of the previous transfer definitions depending on the system we are extending.

The arbiter is obtained from the corresponding previous arbiter by adding two additional registers for every master to sample burst request data. The interface to the master is also extended with two inputs $bst \in \mathbb{B}$ and $bsize \in \mathbb{B}^b$ indicating a request for a burst transfer of size $bsize$. Upon an active req signal from some master, a flag indicating a burst request is set and the $bsize$ input is sampled if the bst input active. When the bus is granted to a master with a pending burst request, the arbiter keeps the $grant$ vector stable for $bsize$ may transfers.

The slave is the same as in the corresponding sequential or pipelined system, thus the transformation is the identity.

Again, the more interesting part is the transformation of the master. The basic idea is again simple: we add a counter for the *sub-transfers* and simulate a sequence of $bsize$ many standard transfers. Thereby, the arbiter correctness ensures that the bus is granted during the complete burst transfer. We specify the transformation in terms of an input transformation $ITrans$ and an output transformation $OTrans$.

The interface from host to master has to be extended with two new signals: $hbst \in \mathbb{B}$ signals that the current transfer is a burst request of size $hbsize \in \mathbb{B}^b$. We also introduce a signal to the host called $bdataupd \in \mathbb{B}$. For write transfers, it signals that the data in $lwdata$ has to be updated und we require the host to update the $wdata$ input when the signal is active. For read transfers, it indicates that new data can be read from the $hrdata$ output in the next cycle. We define $bdataupd$ in detail as part of $OTrans$.

We also have to extend the configuration of the master with three new components to handle burst requests: $bst \in \mathbb{B}$ and $bsize \in \mathbb{B}^b$ are used to sample the corresponding host signals. As the local address register has to be incremented before every burst sub-transfer except the very first, we introduce a flag $bfst \in \mathbb{B}$ indicating the first one. The configuration of a master BM_u is given by $bm_u = \{s,p\}m_u \times (bst, bsize, bfst)$.

In contrast to $ITrans$ for the pipelined master, this transformation is a state-representing extension. Thus, we describe $ITrans$ as a finite state machine, i.e. by a state-transition function $bmi\delta$ and a output function $bmi\omega$. Let $bmi = (rdy, grant, rdata, startreq, hinp)$ denote the inputs to a master with $hinp = (hwr, haddr, hwdata, hbst, hbsize)$ and let $\{S,P\}M$ denote the master which is extended. Moreover, let $done = (bsize = 0)$ and

$$bupd = \begin{cases} rdy \wedge \neg bfst & : PM \\ rdy \wedge (state = dph) & : SM \end{cases}$$

Then $bmi\delta\ ((bst, bsize, bfst), bmi)$ is given by the tuple $(bst', bsize', bfst')$ where

$$bfst' = \begin{cases} 1 & : startreq \wedge \neg\{s,p\}m.busy \\ & \quad \wedge\ hbst \wedge \neg(grant \wedge rdy) \\ 0 & : rdy \wedge grant \\ bfst & : \text{otherwise} \end{cases}$$
$$bst' = \begin{cases} hbst & : startreq \wedge \neg\{s,p\}m.busy \\ 0 & : bupd \wedge done \\ bst & : \text{otherwise} \end{cases}$$
$$bsize' = \begin{cases} hbsize & : startreq \wedge \neg\{s,p\}m.busy \\ bsize - 1 & : bupd \wedge bst \wedge \neg done \\ bsize & : \text{otherwise} \end{cases}$$

Given $nextbst$ as a shorthand for $bupd \wedge bst \wedge \neg done$, the output is given by $bmi\omega\ (bm, bmi) = (bstartreq, bwr, baddr)$ where

$$bstartreq = startreq \vee nextbst$$
$$bwr = \begin{cases} \{s,p\}m.lwr & : nextbst \\ hwr & : \text{otherwise} \end{cases}$$
$$baddr = \begin{cases} \{s,p\}m.laddr + 1 & : nextbst \\ haddr & : \text{otherwise} \end{cases}$$

Given $ITrans$, we can define the state-transition function of a master supporting burst transfers. Let xm for $x \in \{s,p\}$ denote the state components of some burst master bm which are also part of the state of the master to be extended, i.e. $bm = (xm, bst, bsize, bfst)$. Then the state-transition function $bst_M(\{s,p\}\delta_M)$ is define as:

$$bst_M(x\delta_M)\ (bm, bmi) \\ = (xm', bmi\delta\ ((bst, bsize, bfst), bmi))$$

where

$$xm' = x\delta_M\ xm\ (rdy, grant, rdata, bstartreq, \\ bwr, baddr, hwdata) \\ (bstartreq, bwr, baddr) = bmi\omega\ (bm, bmi)$$

Next we specify the output transformation. Except for the $busy$ and $bdataupd$ outputs, the outputs remain the same

as for the previous masters. The signal to update the burst data is given by $bdataupd^t = bupd^t \wedge bst^t \wedge \neg done^t$ where $done$ denotes $bsize = 0$ as before. The $busy$ signal has to be adapted for the case a burst access is in progress such that it remains active during that transfer. We obtain $busy^t = \{s,p\}m.busy^t \vee (bst^t \wedge \neg done)$. For all other outputs, $OTrans$ is just the identity.

Finally, we must specify the correctness predicate. We spilt cases on burst transfers: in case of a non-burst transfer, the correctness predicate is the one from the 'source' system.

In case of a burst transfer, the correctness predicate specifies a sequence of correct single transfers. Let $1 \leq n \leq bsize - 2$ and $1 \leq m \leq bsize - 1$, then it is given by:

$$
\begin{aligned}
&validreq_u^t \wedge hbst_u^t \implies \\
&\ i(u,t) \neq -\infty \wedge btr(i(u,t)).isdata \wedge \\
&\neg hwr_u^t \implies rdata_u^{ta(1)+1} = gm^{tg}(haddr_u^t) \wedge \\
&\qquad\qquad rdata_u^{ta(n+1)+1} = gm^{tg}(haddr_u^t + n) \wedge \\
&\qquad\qquad rdata_u^{td+1} = gm^{tg}(haddr_u^t + bsize - 1) \wedge \\
&hwr_u^t \implies gm^{td}(x) = \begin{cases} hwdata_u^t & : x = haddr_u^t \\ hwdata_u^{ta(m-1)} & \\ & : x = haddr_u^t + m \\ gm^{tg}(x) & : \text{otherwise} \end{cases}
\end{aligned}
$$

The validity is again shown using the correctness of the components. Using the input and output transformation of the master, which are correct by construction, a burst transfer simulates a sequence of 'normal' transfers.

V. Conclusion

We have illustrated our general approach with a common and accessible protocol example (AMBA) and cover two very general features: pipelining and burst mode. A broad variety of on-chip busses support both features. Our initial focus on AMBA also provides a useful comparison with previously published work on this protocol [11], [12] that relied on post-hoc verification. We believe our general approach and also the specific idea we presented for realizing transformations as input and output restrictions will provide a basis for many other optimising transformations.

All definitions have been formalized and theorems have been proven using the Isabelle/HOL theorem prover [18]. Local properties of the finite state machine have been shown using the symbolic model checker NuSMV [19] used via an oracle-based interface [20].

These initial results provide encouraging evidence for the practicability of our approach, but are only a starting point. Our larger scientific hypothesis is that we are able to derive much more complex, verified protocols with a similar refinement approach and modelling framework. Our target is not AMBA, but high-performance protocols approaching the complexity and subtlety of today's industrial implementations.

Much remains to be done to test this hypothesis. We will need more transformations, and systematic methods of applying them supported by automatic proof. We will need to invent a greater range of refinement relations to link different abstraction levels. These will relate atomic, high-level transfers to non-atomic communications spread over space and time at the cycle-accurate, register-transfer level (see [14] for example). Finally, we will need to tackle circuit issues arising from the implementation layer—for example, we expect to develop methods to argue about low-level timing constraints or distributed, asynchronous clocks. As part of this, we expect to identify and characterise the right level of abstraction at which our refinement towards implementation will stop—and that this will be somewhat below the conventional unit-delay or half clock-cycle levels usually assumed in much of today's formal verification research.

VI. Acknowledgements

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

References

[1] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT*, vol. 8, pp. 174–186, Feb. 1968.
[2] ——, *A Discipline of Programming*. Prentice-Hall, 1976.
[3] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, no. 4, pp. 221–227, 1971.
[4] *AMBA Specification Revision 2.0*, ARM, 1999.
[5] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991.
[6] K. L. McMillan, "A compositional rule for hardware design refinement," in *CAV '97*. Springer, 1997, pp. 24–35.
[7] M. Aagaard, B. Cook, N. A. Day, and R. B. Jones, "A framework for microprocessor correctness statements," in *CHARME '01*. Springer, 2001, pp. 433–448.
[8] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *ICCD'92*, 1992, pp. 522–525.
[9] A. T. Eiríksson, "Integrating formal verification methods with a conventional project design flow," in *DAC '96*. ACM, 1996, pp. 666–671.
[10] S. M. German, "Formal design of cache memory protocols in IBM," *Formal Methods in System Design*, vol. 22, no. 2, pp. 133–141, 2003.
[11] A. Roychoudhury, T. Mitra, and S. Karri, "Using formal techniques to debug the amba system-on-chip bus protocol," *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 828–833, 2003.
[12] H. Amjad, "Model checking the AMBA protocol in HOL," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-602, Sep. 2004. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-602.pdf
[13] J. Schmaltz and D. Borrione, "Towards a formal theory of on chip communications in the ACL2 logic," in *ACL2 '06*. ACM, 2006, pp. 47–56.
[14] X. Chen, S. M. German, and G. Gopalakrishnan, "Transaction based modeling and verification of hardware protocols," in *FMCAD '07*. IEEE Computer Society, 2007, pp. 53–61.
[15] F. Müffke, "A better way to design communication protocols," Ph.D. dissertation, University of Bristol, May 2004.
[16] C. Seger, "The design of a floating point unit using the integrated design and verification (IDV) system," in *DCC '06: Participants' Proceedings*, M. Sheeran and T. Melham, Eds., March 2006.
[17] P. Böhm and T. Melham, "Design and verification of on-chip communication protocols," Oxford University Computing Laboratory, Research Report RR-08-05, April 2008. [Online]. Available: http://web.comlab.ox.ac.uk/publications/publication1538-abstract.html
[18] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.
[19] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. R. Marco Pistore, R. Sebastiani, and A. Tacchella, "NuSMV 2: An open source tool for symbolic model checking," in *CAV '02*. Springer, 2002, pp. 359–364.
[20] S. Tverdyshev, "Combination of Isabelle/HOL with automatic tools," in *FroCoS '05*, ser. LNCS, vol. 3717. Springer, 2005, pp. 302–309.