Refinement-Based Formal Verification of Asynchronous Wrappers for Independently Clocked Domains in Systems on Chip

Xiaohua Kong, Radu Negulescu, and Larry Weidong Ying

Microelectronics And Computer Systems Laboratory
Dept. of ECE, McGill University
Montreal, Quebec, Canada H3A 2A7
Tel: (514) 398-2194, Fax: (514) 398-4470
{kong,radu,larry}@macs.ece.mcgill.ca

Abstract. In this paper we propose a novel refinement-based technique to formally verify data transfer in an asynchronous timing framework. Novel data transfer models are proposed to represent data communication between two locally independent clock domains. As a case study, we apply our technique to verify data transfer in a previously published architecture for globally asynchronous locally synchronous on-chip systems. In this case study, we find several race conditions, hazards, and other dangers that were not mentioned in the original publication, and we find additional delay constraints that avoid some of the detected dangers.

1 Introduction

With smaller feature sizes and larger die areas, integrating systems on a chip has become a feasible solution to satisfy unrelenting demands to provide more functionality at higher clock rates. Future on-chip systems designs are expected to involve the integration of multiple subsystems with independently clocked domains, including memory cores and processor cores. An advantageous scheme for realizing communication among clock domains is by means of glue logic circuits that operate in a self-timed manner, i.e., without a clock. Robust glue logic circuits are critical to ensure safe and stable data transfer at high operating speeds.

Synchronizing inputs with a local clock requires arbitration, which has a high probability of metastability at high operating speeds. Pausible clock schemes, proposed in [20], handle metastability by extending the passive phase of the clock long enough for metastability to resolve.

Globally-asynchronous locally-synchronous architectures (GALS), proposed by [33], [8] and [9] among others, show that a system can be partitioned into several independently clocked domains (subsystems) that communicate in a self-timed manner. To isolate each locally-synchronous domain from its globally-asynchronous environment, [1], [8] and [9] introduced an elegant design, called asynchronous wrapper, used to equip each locally-synchronous domain. Asynchronous wrappers serve as controllers for data transfer between individual domains, and deliver a locally generated pausible clock for the synchronous part of circuitry [8].

The interface-based design methodology of [18] proposes to define a system on chip architecture by a high-level block partition on one hand, and by multi-level communication policies on the other hand. Communication refinement was promoted in [18] as a technique for rapid development of system on chip architectures by successive refinement applied to communication policies. In this paper, we show how to formally verify a key step of communication refinement for a GALS architecture, by verifying that a channel configuration with asynchronous wrappers meets a higher-level data-transfer specification that abstracts away the communication protocol used.

Verification of communication refinement is both important and non-trivial. We believe that the adoption of GALS architectures in future on-chip systems design will depend on the extent to which these novel architectures can guarantee safe and stable data transfer, while preserving robustness of the system designs. These concerns can be generally attributed to data and system safety issues. Although the simulations and tests reported in [8] and [9] show that their design works properly, such simulations and tests can only cover a small fraction of all possible configurations of relative delays in asynchronous communication, while we find that various malfunctions may still emerge under different relative delays. Applying formal verification, we find the potential pitfalls and provide delay constraints that permit to size the circuits so that the pitfalls are avoided.

The main challenges in our verification tasks are as follows: we have to combine in uniform models two essentially different synchronization paradigms (edge-triggered synchronous and handshake asynchronous); our data transfer specifications must be kept simple in the presence of non-synchronized clock cycles, which entails, for instance, that data can not be updated on every clock cycle on the sender and receiver sides; and, our specification and verification must be kept simple in the presence of interleavings of datapath and control signals.

In this paper, we: 1) propose new data transition model to represent the implicit relationship between clock and data validity events; 2) construct comprehensive implementation models for the asynchronous wrapper and the asynchronous communication scheme; 3) report several design pitfalls, including hazards in a design, obtained from 3D synthesizing tool, which claimed by [9] to be hazard-free; 4) provide relative timing constraints that were not mentioned by [8] and [9], along with fault diagnosis which indicates that the disregard of these constraints can cause system deadlock or erroneous data transfers.

Formal methods for verifying protocol conversion and refinement have been proposed for instance in [2], [3], [15] and [19]; [2] applies some of these ideas to the design of asynchronous circuits. However, our method is based on transforming high-level specifications that use data validity events into low-level specifications that use signal transition events, whereas the cited previous work only interfaces protocols by means of physical circuits which add latency and buffering. The result of our specification is similar to a CFSM (communication finite state machines) as in [15]. However, [15] does not include specifications in terms of data validity events.

Following [11], our verification method uses metric-free models for systems that rely on relative timing. Compared to timed methods, such as [5] and [10], metric-free verification is less computationally-intensive and integrates more easily within a hierarchical verification framework that permits non-determinism. Metric-free verification methods for relative timing are also used in [17].

2 Preliminaries

In subsection 2.1, we briefly overview the necessary notions from our verification framework, called *process spaces*; for more details, we refer the reader to [11] and [12]. In subsection 2.2., we briefly overview the *active-edge* specifications introduced in [7] and [13], which are used here to simplify the specification of control signals. In subsection 2.3, we briefly overview the asynchronous data communication schemes from [9] that are used in our verification case study.

2.1 Process Spaces

Process spaces are a general theory of concurrency, parameterized by the execution type. Systems are represented in terms of their possible executions, which can be taken to be sequences of events, functions of time, etc., depending on the level of detail desired in the analysis. This permits to trade tractability against precision of analysis by choosing an execution type. Modular and hierarchical verification are supported. If executions are taken to be finite traces (finite sequences of events), close relationships exist to several previous treatments of concurrency, such as [5] and [6].

Let E be the set of all possible executions. A *process* p is a pair (X, Y) of subsets of E such that $X \cup Y = E$. A process represents a contract between a device and its environment, from the device viewpoint. Executions in $X \cap Y$, called *goals*, denoted by **g** p, are legal for both the device and the environment. Executions from outside X, called *escapes*, denoted by **e** p**,** represent bad behavior on the part of the device. Finally, executions from outside Y, called *rejects*, denoted by **r** p, represent bad behavior on the part of the environment. We also use **as** p (accessible) and **at** p (acceptable) to denote X and Y respectively.

Process spaces can be used to build models of circuit behavior in a manner similar to conventional state machines. For an example of the models used in this paper, consider the C-element in Fig.1 (a). If the inputs a and b have the same logical value, the C-element copies that value at the output c; otherwise, the output value remains unchanged. Waveforms are represented by finite sequences of actions corresponding to signal transitions, such as *abcbac* for the waveform in Fig. 1 (b). In this paper, we use the term *trace* to refer to such a sequence of actions. We sometimes indicate that a certain action represents a rising or falling transition, as in a+ b+ c+ b- a- c-.

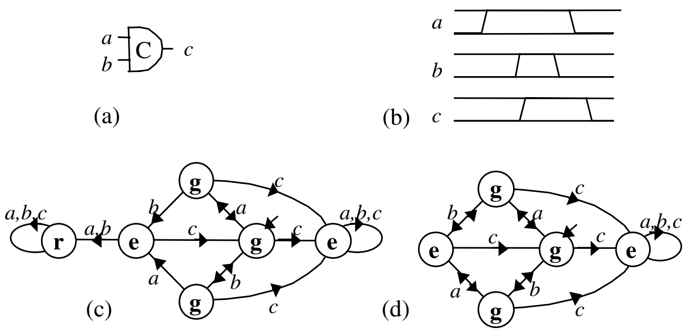


Fig. 1. Example processes: (a) The C-element symbol we use; (b) Waveform; (c) Hazard-intolerant model; (d) Inertial model.

If all signals start low, the C-element can be represented by the process in Fig. 1 (c), where **r**, **g**, and **e** stand for reject, goal, and escape. Illegal output events lead to an escape state with self loops on all subsequent events, call it a *permanent escape*, and illegal input events lead to a reject state that cannot be left either, call it a *permanent reject*. The state where *ab* leads is also marked **e**, making it illegal for the device to complete its operation by stopping there.

The model in Fig. 1 (c) is a *hazard-intolerant* model. There are variations of the CMOS cell models, because, in the presence of hazards, the behavior of a CMOS cell is not fully standardized. A hazard is a situation where an output transition is enabled and then disabled without being completed. For example, execution *abb* is a hazard for the C-element in Fig. 1. Hazard-intolerant models simply require the environment to avoid hazards, by stating that each execution that includes a hazard will lead to a permanent reject. The model in Fig. 1 (d) is an *inertial* model. Inertial models ignore hazards by filtering out input pulses that are shorter than the delay of the gate.

Our processes can be used to model not only gates or cells, but also relative timing assumptions of the following form:

$$D\,(b_1\,b_2\,...\,b_n)\ >\ D\,(a_1\,a_2\,...\,a_m)$$

where a_1, ..., a_m, b_1, ..., b_n are events such that a_1 is the same as b_1, and the Ds are the durations of the chains of events. Such a constraint, called a *chain constraint* [14], enforces that the b chain of events will not be completed before the a chain (unless one of the a or b actions involved occurs out of order).

Treating constraints as processes rather than linear inequalities permits us to deal with cases of deadlock and non-determinism, where the inequalities might not apply. Chain constraints can be implemented by transistor sizing. The absence of numerical information in the process models for chain constraints leads to more efficient verification using existing tools for non-timed analysis. Metric-free verification under relative timing constraints was first presented in [14] and [11].

In this paper, we only use the following operations and conditions on processes:

- *Refinement* is a binary relation, written $p \sqsubseteq q$, meaning "process q is a satisfactory substitute for process p".
- *Product* is a binary operation, written $p \times q$, yielding a process for a system of two devices operating "jointly". Product is defined by $\mathbf{as}\,(p \times q) = \mathbf{as}\,p \cap \mathbf{as}\,q$ and $\mathbf{g}\,(p \times q) = \mathbf{g}\,p \cap \mathbf{g}\,q$.
- *Robustness* is a unary predicate on processes, written R_E, defined by $\mathbf{r}\,p = \varnothing$, which represents a notion of absolute correctness: the device is "fool-proof" and can operate in any environment.
- *Reflection*, written $-\,p$, defined by $\mathbf{as}\,(-\,p) = \mathbf{at}\,p$ and $\mathbf{at}\,(-\,p) = \mathbf{as}\,p$, represents a swap of roles between environment and device.

Refinement is reflexive, transitive, and antisymmetric. Product is commutative, associative, and idempotent. Furthermore, for processes p, q, and r,

$$p \sqsubseteq q \;!\; \quad p \times r \sqsubseteq q \times r.$$

These properties suffice to break a verification problem into several layers of partial specifications, and each layer into several modules, and to verify only one module at a time instead of verifying the overall problem in one piece.

Manipulations of finite-word processes are implemented by a tool called FIREMAPS [11] (for finitary and regular manipulation of processes and systems). This tool uses a BDD library [5] which offers basic routines for manipulating large Boolean functions. FIREMAPS implements the process space operations and conditions mentioned above, and has built-in constructors for hazard-intolerant, and inertial models, and for chain constraints. In addition, if refinement does not hold, FIREMAPS can produce a *witness execution* that pinpoints the failure. Such witness executions are used for fault diagnosis.

2.2 Active-Edge Specifications

Most digital circuit components are designed to synchronize on edges of their synchronization signals. Often, only one set of edges (rising or falling) of a synchronization signal are taken into account; such edges are called *active*. The other set of edges, called *passive*, are not used for synchronization. Correspondingly, simpler state-machine representations can be obtained by only referring to the active edges for control signals.

We use the term *active-edge specification* to refer to processes that represent circuit components by ignoring passive edges. For example, as illustrated in [13], the role of the clock signal in an edge-triggered flip-flop can be described by an active-edge specification. We refer to [13] on how to build active-edge specification by a "semi-hiding" operation, and to both [7] and [13] on how to reconstruct full specifications from active-edge specifications.

We use the term *transition-event specification* to denote a process in which both rising edges and falling edges of a signal are specified in the executions. Examples of transition-event models including their specifications presented in subsection 2.1, where each signal transition is represented by an occurrence of the corresponding action.

2.3 Asynchronous Communication

An alternative to clocking for digital circuits is to use handshake protocols to ensure coherent data communication between modules. In such protocols, data communication is synchronized by request and acknowledge signals. This alternative is particularly convenient for communication in heterogeneous designs with modules that have different timing schemes.

The design in [9] uses an early single-rail four-phase protocol [16], illustrated in Fig. 2. In this protocol, data should be guaranteed to be valid between the active edges of the request and acknowledge signals (bundled data). As an example, Fig. 2 shows an early four-phase handshake protocol for a push data channel [16], which is used in the asynchronous wrapper configuration presented in [9].

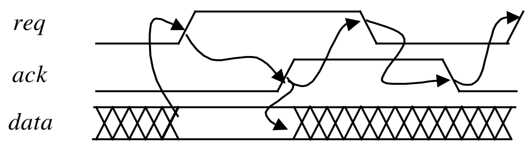
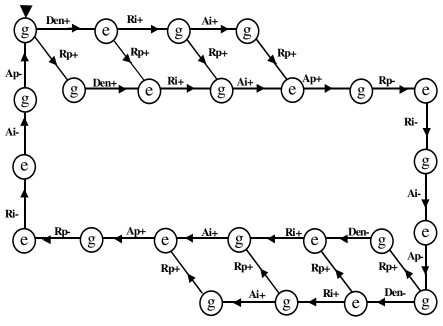


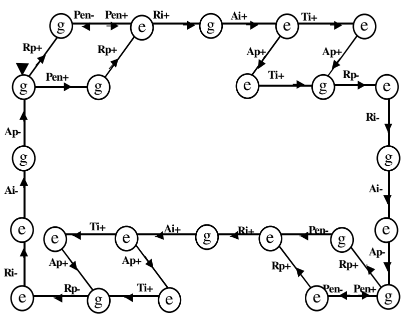
Fig. 2. Four-phase handshake protocol with early data valid scheme

Asynchronous wrappers in GALS architectures provides an interface between locally-synchronous domains and a globally-asynchronous environment. Data communication among independently clocked domains uses a four phases handshake protocol as illustrated in Fig. 2. Arbitration is required to synchronize the handshake signals and associated data with one of the clocked domains. The asynchronous wrapper circuits proposed in [1] and [9] attempted to realize failure-free communication in presence of metastability by performing arbitration between local clocks and handshaking control signals.

An asynchronous wrapper is composed of one pausible clock generator for each independently clocked domain, one input port and one output controller for each data port, and several other control ports. Two types of control ports proposed by [12], *Demand-type* (*D*-port) and *Poll-type* (*P*-port), are employed to control data transfer between two clock domains in a self-timed manner. Fig. 3 illustrates our specifications for a *D*-input port, *P*-input port and *D*-output port from [9]. These specifications were built by us on the basis of informal explanations, extended-burst-mode specifications [21], and waveforms provided by [9]. The only point where we depart from [9] in these specifications is by allowing retraction of the *Pen* signal in Fig. 3 (b), because this retraction actually occurs in the channel configuration shown in [9] if the clock domain that generates the *Pen* signal has a significantly higher frequency than the other clock domain involved in the data transfer. No such retraction is needed for the



(a)



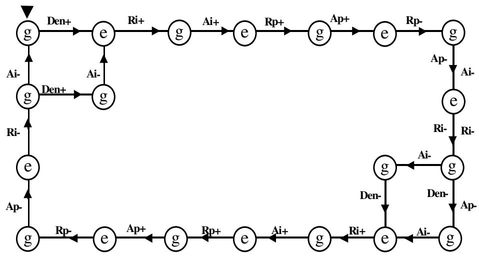
(b)

$$Ri = RpRi + \overline{Den}\,Z_0 + Den\,\overline{Ap}\,\overline{Z_0}$$
$$Ap = RpAi + AiAp$$
$$Z_0 = \overline{Rp}\,Z_0 + \overline{Ai}\,Z_0 + Den\,\overline{Rp}\,Ap$$

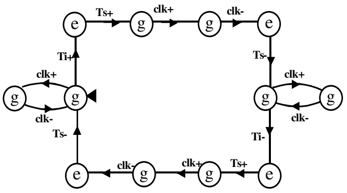
(c)

$$Ri = RpRi + Rp\,\overline{Pen}Ti + RpPen\,\overline{Ti}$$
$$Ti = AiPen + \overline{Ai}Ti + \overline{Ri}Ti$$
$$Ap = Ai$$

(d)



(e)



$$Ts = clkTs + \overline{Ti}Z_0 + TiZ_0$$
$$Z_0 = clkTi + TiZ_0 + \overline{clk}Z_0$$

(f)

Fig. 3. Asynchronous wrapper control ports: (a) *D*-input port specification; (b) *P*-input port specification; (c) Boolean functions of *D*-input port; (d) Boolean functions of *P*-input; (e) *D*-output port specification (f) Translator specification and its implementation

Den signal, because D-type ports block their local clocks. The Boolean functions shown in Fig. 3 are the synthesis results reported in [9].

The primary difference between *D*-ports and *P*-ports is that after an enable signal event, a *D*-port will fire a request to pause the clock before next rising clock edge, whereas a *P*-port does not pause the clock after an enable signal event until it receives corresponding events from handshake line.

Local clocks are generated by ring oscillators with arbitration [20]. A clock pause request will shift the start of the next clock cycle and stretch only the low phase of the clock. Competition between clock pause request *Ri* (see Fig. 4) and clock restart request is arbitrated by a four-phase mutual exclusion element. If both requests arrive simultaneously, then mutual exclusion element will make a non-deterministic selection to pick up one, while guaranteeing that the grant outputs are mutually excluded at all

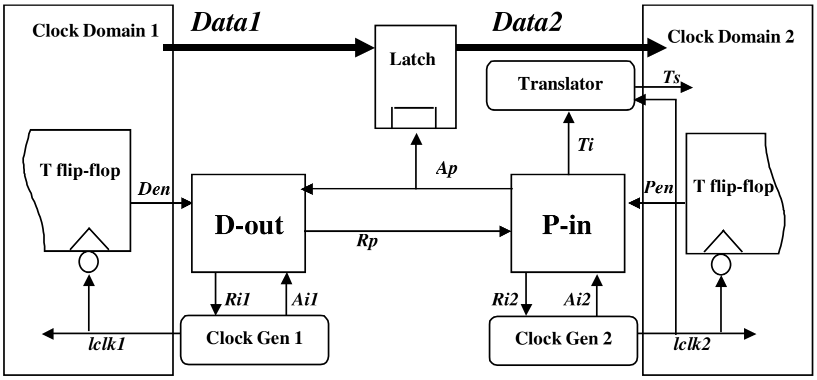


Fig. 4. Data channel between two independently clocked domains

times. Since the *P*-port doesn't have a predetermined clock cycle to transfer data, an extra control unit called translator is required to synchronize data transfer from the asynchronous wrapper to the receiver side of a local clock domain and provide the receiver correct timing to sample the available data at its data input.

Based on different data transfer schemes, several channel configurations can be obtained from various combinations of *D*-type and *P*-type control ports [9]. Following [9], we select *D*-type output with *P*-type input configuration as our verification case study. Following [8] and [9], Fig. 4 shows this control ports configuration. A mutual exclusion element is present in each clock generator.

A data transfer cycle is started by *Den+*. With the constraint reported by [9], a *D*-output clock pause request *Ri1+* should reach the clock generator within half of its clock cycle. After *Ai1+*, clock1 (*lclk1*) remains low and *Rp+* is issued. Clock2 (*lclk2*) will be stopped after both *Rp+* and *Pen+* are fired. *Rp+* is acknowledged by *Ap+*. Clock2 can restart anytime after *Rp-*, whereas clock1 is only allowed to restart after *Ap-*, which indicates the completion of the handshake operation cycle. A latch is inserted in the data path in order to decouple send and receive operations, and ensures availability of stable data at its output when the latch is in opaque state. Two T flip-flops are added as next-state logic in each clock domain, to obtain hazard-free enable signals at their outputs. The incoming signals of the T flip-flops are not necessarily hazard-free.

3 Data Transfer Modeling

To verify correctness of data exchange, we build a high-level specification of data transfer, we convert the high-level specification to a low-level specification by automated means, and check refinement of the low-level specification by the implementation models of the asynchronous wrappers. Essentially, communication among locally-synchronous blocks is implemented by data lines bundled with control signals. We expect the specification should:

- Include features of both control path and data path.
- Be sufficiently precise so that it can support automated verification.
- Be as simple as possible, so that it can be easily developed and understood by designers.
- Be sufficiently general so that it can be easily mapped to communication refinement.

We introduce a new technique to build data transfer specifications independently of the synchronization scheme, and we apply this technique to our verification case study. Although the examples we use in this paper focus on data transfer between two independently clocked domains, the same data transfer specification modeling technique can be applied in a more general asynchronous context.

3.1 Data Transfer Specifications

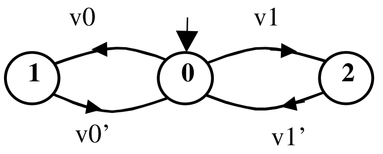
A robust data transfer scheme requires that, under any circumstance, data issued by the sender should be received correctly by the receiver after a certain delay. The notion of correctness used here can be decomposed into the following aspects:

- Data integrity: received data preserves its original sending value.
- Stream integrity: no data items are lost or duplicated during data transmission, and the order of data items is preserved through the transfer.

As modeling stream integrity would require numerous states to represent data and control signal interleavings in a transition-event representation, we use instead a fictitious data event called *validity event* in our high-level specifications. Our technique abstracts away any irrelevant transition events and only considers data events triggered at active edges of clocks. This model fits nicely into a communication refinement paradigm, by permitting to isolate data validity events from the particulars of the synchronization signals used.

A validity-event data-transfer specification incorporates the following assumptions:

- After a control signal is fired by the sender, there is a data validity event at the input port of the data channel.
- After data is properly sampled by the receiver, there is a data validity event at the input port of the receiver.
- Data integrity is preserved in the data transmission, for instance, for each valid input “0” (or “1”) there is a valid output “0” (“1”), and *vice versa*.
- Stream integrity is preserved in the data transmission.



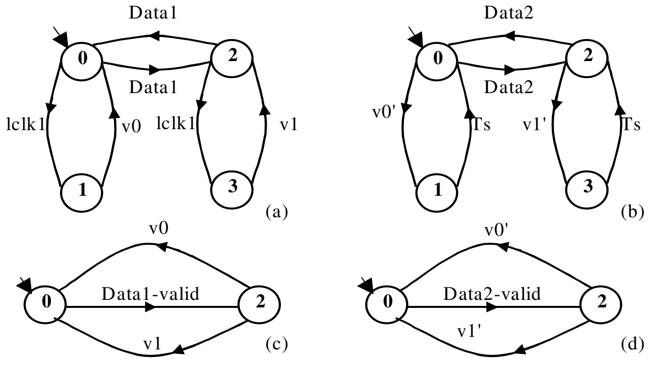
Drawing conventions:
- The states shown are goals.
- Missing *v0* and *v1* events lead to permanent rejects.
- Missing *v0'* and *v1'* events lead to permanent escapes.
- Other actions are ignored and have self-loops at each state.

Fig. 5. Propagation of validity events

For the asynchronous wrapper, the high-level data-validity specification is the process shown in Fig. 5. The "0" data validity event *v0* is propagated as *v0'* to the end of data channel. Same applies for the "1" data validity events.

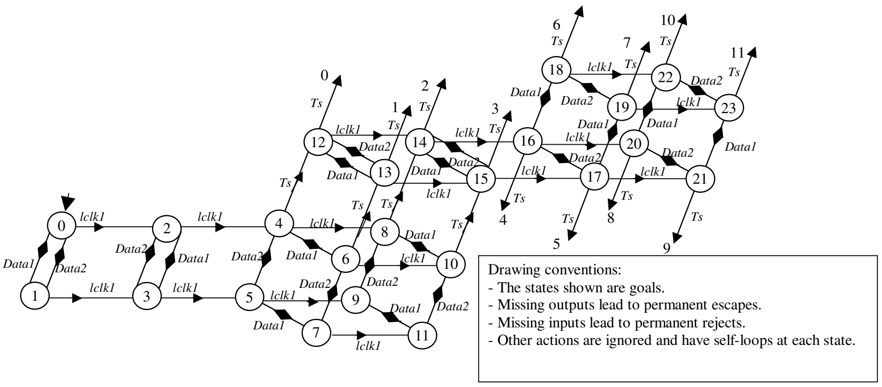
3.2 Fusion Processes

In order to transform the high-level specification of Fig. 5 into a full-blown transition-event specifications, we introduce *fusion* processes to "glue" validity events, transition events, and control events (e.g. clock) which trigger the validity events. Such processes effectively force glued events to occur simultaneously by forbidding other events from occurring in between. For instance, Fig. 6 (a) illustrates a fusion process where the data transition event *Data1* is fused with its validity events *v0* and *v1* by control event *lclk1*, which is the active edge of a local clock. Note that signal *Data1* can toggle arbitrary in a clock cycle, while the validity events are related to the logical level that signal *Data1* has right before active-edge of *lclk1*: If *lclk1* comes when *Data1* is low, validity event *v0* will be issued and there will be no another validity event until the next active edge of *lclk1*, though *Data1* might keep changing between two active clock events. Same as Fig. 6 (a), Fig. 6(b) fuses the transition events of *Data2,* with the active edges of control signal event *Ts* and with corresponding



Drawing conventions:
- The states shown are goals.
- Missing transitions on the actions of each process lead to permanent escapes.
- Other actions are ignored and have self-loops at each state.

Fig. 6. Example of Fusion Processes (a). Fusion of Data1; (b) Fusion of Data2; (c) Invariant fusion of Data1; (d) Invariant fusion of Data2



Drawing conventions:
- The states shown are goals.
- Missing outputs lead to permanent escapes.
- Missing inputs lead to permanent rejects.
- Other actions are ignored and have self-loops at each state.

Fig. 7. Specification constructed from fusions

validity events. Fig. 6(c) and (d) are fusion processes to glue the validity events with a higher-level data-invariant validity event, for the case that where one data-invariant validity event is required to represent either of the lower-level validity events in analysis.

We present in Fig. 7 a state diagram for the process obtained by applying fusion process to "glue" both *data1* and *data2*. To obtain the full-blown transition-event specification, we first compute the product of the high-level specification in Fig. 5 with the fusion events in Fig. 6 (a) and (b), then by hiding the high-level data-validity events. In Fig. 6, only the active edges of the clock signals are represented. Further, we transform the active-edge signals into transition-event signals using the procedures from [7] and [13]. The result of these transformations is the full specification of signal transitions shown in Fig. 7. These transformations are implemented in FIREMAPS.

4 Case Study

In subsection 4.1, we present the construction of asynchronous wrapper models. In subsection 4.2, we discuss the refinement-based verification procedures. Finally, we report our verification results and analysis in subsection 4.3.

4.1 Models Construction

We assume the initial states for each component with all signals are low. The process space specifications for each asynchronous wrapper port type are as shown in Fig. 3. To obtain hazard-free data (stable data) in the design from [8] and [9], the incoming data for asynchronous wrapper are registered before being put onto data channel. Accordingly, a D flip-flop model is included to the data path in our asynchronous wrapper model and validity event data can be obtained at the output of D flip-flop.

To describe the edge-triggered components (T, D), as well as the latch component, we use the active-edge specification models of [7] and [13]. For the lowest-level equations in Fig. 3, we built hazard-intolerant models in FIREMAPS, assuming each equation is implemented by a complex gate (a single standard Boolean gate). The data transfer model of the asynchronous wrapper is constructed based on validity events (see section 3). Fig. 7 shows the resulting model. The translator specification is based on the state machine shown in [8] for that component. The four-phase mutex used to implement clock generation as shown in [8] and [9] was modeled by its standard transition-event process. *D*-output port timing constraint reported at [9] is included in our model, and is built in the form of chain constraint (see subsection 2.1) to represent this timing restriction in design.

4.2 Refinement-Based Verification Cycle

Our first refinement-based verification cycle is started by adding the *D*-output port delay constraints reported in [9] to our asynchronous wrapper's intermediate specification model, and checking the refinement with its high-level specification. FIREMAPS checks refinement and diagnoses the dangers detected in the implementations by providing a set of witness executions that constitute counter-examples to refinement.

We repeat the verification cycle by adding new relative timing constraints to avoid the witness executions detected at previous verification cycles. Relative timing constraints are found through a detailed analysis of asynchronous wrapper design with the help of witness executions obtained from FIREMAPS. These cycles repeat until FIRMAPS eventually reports that the refinement condition is satisfied.

4.3 Verification Results

To check the claim in [9] that hazard-free implementation of port controller could be obtained by synthesizing its extended-burst-mode specification by the 3-D synthesis tool, we verify the individual control components *D*-input control port and *P*-input control port against the hazard-intolerant models of their equations from [9].

Subsequently, we present the most significant of our detected relative timing constraints. We explain the problems indicated by the counter-examples to refinement that are obtained by FIREMAPS when one of the respective constraints is not included, but all the other constraints are included. This shows how ignorance of each constraint in the asynchronous wrappers design may lead to system failures and improper data manipulation under particular timing conditions.

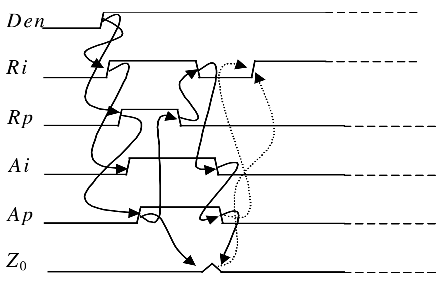


Fig. 8. Illustration waveforms (drawn) of *D*-input port hazard

Control Ports Verification. Here we verify the refinement relationship between specification and implementation for *D*-input ports and *P*-input ports (Fig. 3), respectively. Both *D*-type and *P*-type control ports are used to control data transfer between two independently clocked domains in asynchronous wrappers. To ensure robust data transfer under fewer delay constraints, hazard-free control port circuitry is preferred. Our verification results show that the refinement relationship does hold for *P*-input, but does not hold for *D*-input. In other words, the implementation of the *P*-input is hazard-free, while a hazard exits in the implementation of the *D*-input.

Assuming low initial states of all voltage signals, FIREMAPS reports a witness execution: *Den*+ *Ri*+ *Rp*+ *Ai*+ *Ap*+ *Rp*- *Ri*- *Ai*- *Ap*- *Ri*+ for *D*-input (Fig. 3(c)), showing that a race condition exists at Z_0. Notice that: starting initially with *Den*+, after *Rp*-, with certain propagation delay, Z_0 should rise (Fig. 8). However, if the delay of *Ri*- to *Ai*- (clock pause releasing acknowledge) plus the propagation delay of *Ai*- to *Ap*- is less than the propagation delay of *Ap*+ (previous *Ap* event) to Z_0+, then the Z_0+ event will be cancelled, causing *Ri* to rise again. Fig. 8 depicts the corresponding trace waveform (drawn). Further, since there are no clock events to trigger the T flip-flop, no *Den* events will be generated. Therefore, the corresponding local clock will be stopped indefinitely, and this stopping effect may ripple through other clock domain since no data acknowledge will be fired subsequently by the local domain, thus causing system stall.

We further find that the implementation of the translator block does not refine its specification. Please refer to Fig. 3 (f) for detailed specification and implementation description of the translator (after Fig. 5 in [8]). From an initial state where all signals are low, a *Ts*+ event cannot occur after *Ti*+ is fired. This consequently halts events that are supposed to be fired after *Ts*+. If there are no *Ts* events to trigger the sampling action at the receiver side, no data transfers will occur. This flaw can be fixed by re-synthesizing the state-graph specification in Fig. 3 (f).

Overall Data Transfer Verification. Here we verify a step of communication refinement by checking whether our data-transfer specification in Fig. 7 is satisfied by the channel configuration in [9]. At the present stage, this part of our verification is based on safety models, which only detect the presence of invalid events. Further investigation will be needed to verify liveness properties by using stronger specifications. By applying our verification techniques, we detect several relative timing constraints that were not reported by the designers. To simplify our presentation of results, we only refer to the validity events data1-valid and data2-valid below, instead of any data event. Relative timing constraints are represented as chain constraints. Signal labels refer to Fig. 4.

1. ***D(Ti+ Ts+ lclk2+ lclk2- Ts-) < D(Ti+ Ti-).*** The *Ts*+ event, which should indicate to the receiver block that data has been received, should be fired and become stable within two consecutive *Ti* events. In other words, the delay from *Ti*+ to *Ts*+ should be less than half the period of clock 2. Failing to satisfy this constraint might lead to data being sampled twice at the receiver side, which leads to erroneous duplication of data items. The worst-case scenario is where every data item is duplicated at the receiver side. In [12], there are no mentions of this danger for duplication. Even though the duplication of items might be fixed inside the receiver block by another level of the communication protocol, the duplication would still undesirable because the computation tasks for receiver would be doubled.

2. ***D(Ai2+ Ti+ Ts+) < D(Ai2+ Ap+ Rp- Ai2- lclk2+).*** The delay from *Ai2*+ (which is to acknowledge the pausing of clock2) to *Ts*+ (which is to indicate the receiver the arrival of data) should be less than the delay from *Ai2*+ to restart the clock2. Otherwise, the receiver will not sample the available data at its data input, due to absence of a triggering event *Ai2*+; moreover, the data which was supposed to be sampled by the receiver will be flushed away by the next incoming data by restarting of clock2. In this situation, the data loss will be permanent and unrecoverable.

3. ***D(Den+ Rp+ Ap+) < D(Den+ data-valid).*** Data should be put at the input port of the latch before the latch switches from transparent to opaque; otherwise, before data getting stable, improper data states will propagate through latch and be sampled by receiver. This constraint sets a delay time bound for the latch to switch its state.

4. ***D(Rp+ Ap-) < D(Rp+ Ts+ lclk2+).*** A *Ts*+ event should be issued later than the *Ap*- event to ensure a stable and valid data at the output of the latch, which was triggered by *Ap*- and switched to opaque state already; otherwise, the *Ts*+ event will trigger the receiver to sample a data item which is not guaranteed to be correct.

5. ***D(Pen+ Ai2+) < D(Pen+ lclk2+).*** The delay path from the *P*-input enabling signal *Pen+,* to the clock pausing acknowledge signal, *Ai2,* should take less time than the issuing of the next *lclk2* event; otherwise, the next *P*-input enabling signal can be ignored as the result of a race condition.

6. ***D(Pen+ Rp- Ri2+ lclk2+ lclk2- Pen-) < D(Pen+ Rp+ Ri2+ lclk2+ lclk2- Ai2+ Ti+ Ri2-).*** The relative timing interval between *Rp*+ and *rclk2*+ is arbitrary. If *Rp*+ is issued closely enough to *lclk2*+, then, *Ri2*+, which was supposed to be triggered by both *Rp*+ and a *Pen* event (*Pen+/-)* can not win the arbitration over *lclk2*+. Therefore, *lclk2* will not be paused immediately after the arrival of *Rp*+, the next clock event *lclk2*- will be fired, and, further, *Pen* will be reset. Thus, a *Ti* event would be canceled, before the *Ts* is set to high. Meanwhile, the acknowledge *Ai2* will still be sent to the D-port, though no data is sampled by the receiver. Moreover, if the implementation of the translator is not totally hazard-free, *Ts* will quickly return to low if *Pen-* is issued right after a *Ts+* event. If the clock2 pause request (*Ri2)* cannot withdraw before *Pen-* comes, *Ti* will be reset again and still might affect the state of *Ts*. We add the above constraint, which implies the delay from *lclk2-* to *Pen-* is longer than the delay from *Ai2+* to *Ri2-*, so that the data item will not be lost during transfer.

5 Conclusion

In [9], the authors introduced an elegant design, called asynchronous wrapper, used to interface locally-synchronous domains to a global asynchronous handshake environment. This design was reported in [9] to be synthesized by an asynchronous synthesis tool, called 3D, and was considered in [9] to be hazard free. In addition, [9] reported several delay constraints to avoid race conditions in their circuits.

We introduce a novel technique to construct and verify specifications for data communication between independent clock domains. We apply our technique to the asynchronous wrapper design in [9], and we find several hazards, additional race conditions, and other faults. We indicate the corresponding observable failures.

Hazards and race conditions often escape detection by non-exhaustive methods, such as simulation, prototyping, and even testing of a few fabricated circuits; given the manufacturing variations and parameter fluctuations present in large systems on chip, only a minor percentage of the fabricated circuits can have the same delay configuration as the tested or simulated ones. We believe that the availability of formal verification techniques for communication refinement is necessary for the creation of robust GALS architectures and for the subsequent acceptance of such architectures by the industry.

Acknowledgments. We thank Mark De Clercq, Mark Greenstreet, and the anonymous CHARME reviewers for suggestions of improvement of the presented material. We are grateful for financial support from NSERC, FCAR, CFI, MEQ, and McGill University.

References

1. Bormann, D., Cheung, P.: "Asynchronous Wrapper for Heterogeneous Systems." In Proc. Int. Conf. Computer Design (ICCD), Oct. 1996.
2. Brown, G., Luk, W., O'Leary, J.: "Retargeting a Hardware Compiler Proof Using Protocol Converters." In Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC), pp: 54-64, 1994.
3. Calvert, K.L.; Lam, S.S.: "Formal methods for protocol conversion." In IEEE Journal on Selected Areas in Communications, Jan. 1990, pp: 127 –142.
4. Chapiro, D. M.: Globally-Asynchronous Locally-Synchronous Circuits. PhD thesis, Stanford University, U.S.A., Oct. 1984.

5. Dill, D. L.: Trace theory for Automatic Hierarchical Verification of Speed-Independent Circuits. (An ACM Distinguished Dissertation.) MIT press, 1989.
6. Hoare, C. A. R.: Communicating Sequential Processes. Prentice Hall, 1985.
7. Kong, X., Negulescu, R.: "Formal Verification of Pulse-Mode Asynchronous Circuits." In Proc. Asia and South Pacific Design Automation Conference (ASP-DAC), Jan. 2001.
8. Muttersbach, J,. Villiger, T,. Fichtner, W.: "Practical Design of Globally-Asynchronous Locally-Synchronous Systems." In Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC), Mar. 2000.
9. Muttersbach, J,. Villiger, T,. Kaeslin, H,. Felber, N,.Fichtner, W.: "Globally-Asynchronous Locally-Synchronous Architectures to Simplify the Design of On-Chip Systems." In Proceedings of ASIC/SOC Conference, pp. 317 -321, 1999.
10. Myers, C.J.; Rokicki, T.G.; Meng, T.H.-Y: "POSET timing and its application to the synthesis and verification of gate-level timed circuits." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume:20, pp: 769 -786, June 1999.
11. Negulescu, R. : Process Spaces and Formal Verification of Asynchronous Circuits. PhD thesis, University of Waterloo, Canada, 1998.
12. Negulescu, R.: "Process spaces." In Proc. Int. Conf. on Concurrency Theory (CONCUR), pp. 196-210, Aug. 2000.
13. Negulescu, R., Kong X.: "Semi-hiding Operators and the Analysis of Active-Edge Specifications for Digital Circuits." In Proc. Int. Conf. on Application of Concurrency to System Design (ICACSD-2001), Apr. 2001.
14. Negulescu, R., Peeters, A.: "Verification of speed-dependences in single-rail handshake circuits." In Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems, pp. 159-170, 1998.
15. Okumura, K.: "Generation of proper adapters and converters from a formal service specification." In Proc. of Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The Multiple Facets of Integration, pp: 564 –571, 1990.
16. Peeters, A. M. G.: Single-Rail Handshake Circuits. PhD thesis, Eindhoven Univ. of Tech nology, Eindhoven, The Netherlands, Jun. 1996.
17. Peña, M. A., Cortadella, J., Kondratyev, A. and Pasor, E.: "Formal verification of safety properties in timed circuits." In Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (ASYNC), pp 2-12, 2000.
18. Rowson, J.A., Sangiovanni-Vincentelli, A.: "Interface-Based Design." In Proceedings of the 34th Design Automation Conference (DAC), pp.: 178 -183, 1997.
19. Tao, Z.P., v. Bochmann G., Dssouli, R.: "An efficient method for protocol conversion." In Proc. Int. Conf. on Computer Communications and Networks, pp. 40-47, 1995.
20. Yun, K. Y., Donohue, R. P.: "Pausible clocking: a first step toward heterogeneous systems." In Proc. Int. Conf. Computer Design (ICCD), pp. 118 –123, 1996.
21. Yun, K. Y., Dill, D. L.: "Automatic synthesis of extended burst-mode circuits: Part I and Part II." IEEE Transactions on Computer-Aided Design, vol. 20, no. 2, pp. 101-132, 1999.