FORMAL VERIFICATION OF PEEPHOLE OPTIMIZATIONS IN ASYNCHRONOUS CIRCUITS

Xiaohua Kong, Radu Negulescu
Montreal, Quebec, Canada H3A 2A7
Department of Electrical and Computer Engineering, McGill University
E-mail: {kong,radu}@macs.ece.mcgill.ca

Abstract This paper proposes and applies novel techniques for formal verification of peephole optimizations in asynchronous circuits. We verify whether locally optimized modules can replace parts of an existing circuit under assumptions regarding the operation of the optimized modules in context. A verification rule related to assume-guarantee and hierarchical verification is presented, using relative timing constraints as optimization assumptions. We present the verification of speed-optimizations in an asynchronous arbiter as a case study.

Keywords: Peephole optimization, formal verification, peephole rule, relative timing, process spaces

1. INTRODUCTION

The dramatic increase of integrated circuit complexity raises the need for guarantees of correctness early in the design flow, instead of leaving such guarantees to testing after the design is completed. Presently, simulation is widely used to provide guarantees of correctness of designs. Unfortunately, simulation typically covers only a small subset of system behaviors, especially for asynchronous circuits, where several interleavings of signal transitions must be taken into account. In contrast, formal verification checks the correctness of a system under all possible signal interleavings. In this paper, we verify whether an implementation meets the specification by checking a refinement partial order on processes; we refer to this verification approach as *refinement checking.*

Typical verification problems for multi-component systems are PSPACE-complete (see e.g. [17]). Several *structured verification* approaches can be used to alleviate state explosion in refinement checking by breaking the verification of a large system into several sub-tasks, and establishing refinement individually for each subtask. One structured verification approach is *hierarchical verification,* where circuit descriptions are provided at several different levels of abstraction. At each level, the circuit is treated as an interconnection of modules. Refinement is checked only between successive levels: the higher level is treated as specification and the lower level is treated as implementation. The description of circuit behavior at a level between the top and bottom levels is called an *intermediate specification.* Hierarchical verification can reduce computational costs if the verification tasks at successive levels involve fewer components than the overall verification task. Another structured verification approach is *assume-guarantee* reasoning, which is essentially an induction-like argument that breaks the circularity of mutual reliance of components in a system.

The application of hierarchical verification depends on finding suitable abstractions for components, which is difficult for systems that have poor modularity. For example, peephole optimizations in digital circuits often introduce additional dependencies between the optimized submodules and the rest of the system, making it difficult to obtain suitable abstractions for the application of hierarchical verification.

Correspondingly, we propose verification techniques that adapt assume-guarantee and hierarchical verification rules to the verification of peephole optimizations. Using relative timing constraints as optimization assumptions, we can verify a circuit after peephole optimizations by: 1) verifying the corresponding circuit before optimization, while it has high modularity; 2) verifying the local replacement between circuit before optimization and circuit after optimization. If proper constraints are given, even optimizations that change the interface of a submodule can be verified quickly based on the verification of the circuit before optimization.

The framework for formal verification used in this paper is process spaces [19, 20]. Our hierarchical verification is related to the methods in [5] and [6]. Assume-guarantee rules are addressed in [5], [4], [15], [10], [11]. We extend the application of such rules by introducing optimization assumptions in verification. A distinctive point of our approach is that arbitrary processes can be used as optimization assumptions, regardless of connectivity; the choice of such processes affects the result mainly through computational costs, as explained in Section 3. In particular, our optimization assumptions can be processes representing relative timing constraints. Also, our proposed verification technique does not rely on induction properties; the proof of Theorem 1 is given without reference to the structure of executions.

2. PRELIMINARIES

2.1 Process Spaces

Process spaces [19, 20] 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. In this paper, executions are taken to be traces (sequences of events); for trace executions, close relationships exist to several previous treatments of concurrency, such as [6] and [9].

Let E be the set of all possible executions. A *process* p is a pair (X, Y) of subsets of E such that $X \ll Y = E$. A process represents a contract between a device and its environment, from the device viewpoint. Executions in X? 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.

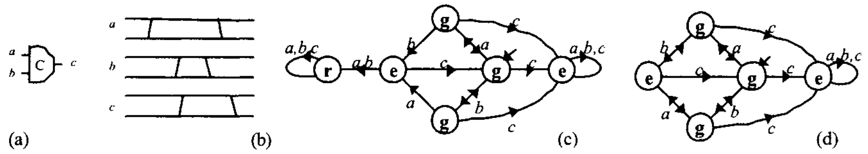

Figure 1. Exampleprocesses: (a) The C-element symbol we use; (b) Waveform; (c) Hazard-intolerantmodel; (d) Inertial model.

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-.

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.

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

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

where $a_1, \dots, a_m, b_1, \dots, b_n$ 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* [17], 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 presented in [19] and [17].

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

?? *Refinement* is a binary relation, written $p \lesssim q$, meaning "process q is a satisfactory substitute for process p".

?? *Product* is a binary operation, written $p ? q$, yielding a process for a system of two devices operating "jointly". Product is defined by **as** $(p?q)$= **as** p ? **as** q and **g** $(p?q)$ = **g**p ? **g** q.

?? *Robustness* is a class of processes, written R_E, so that process p is in R_E if and only if **r** p = ? . Robustness represents a notion of absolute correctness: the device is "fool-proof" and can operate in any environment.

?? *Reflection,* written $-p$, defined by **as** $(-p)$ = **at** p and **at** $(-p)$ = **as** p, represents a swap of roles between environment and device.

In process spaces, refinement is reflexive, transitive, and antisymmetric; product is commutative, associative, and idempotent. Furthermore, for processes p, q, and r,

$$p \underline{\ll} q \quad ? \qquad p\ ?\ r \underline{\ll} q\ ?\ r.$$

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 the overall problem in one piece.

Product admits an identity element, which is the process ? defined by g ? = E. Note that a process is robust if and only if that process refines ? .

Manipulations of finite-word processes are implemented by a BDD-based tool called FIREMAPS [17] (for finitary and regular manipulation of processes and systems). 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 Peephole Optimizations in Asynchronous Circuits

By *peephole optimizations* we mean local changes in circuit sub-modules that do not affect the rest of the circuit or the operation of the circuit as a whole. This sometimes involves changes in the interface of the respective sub-module to take advantage of signals available in other modules.

Peephole optimizations are often performed after high-level synthesis with significant gains. In [7], a peephole optimization step after synthesis by the methods in [1] and [3] is shown to produce gate count improvements up to a factor of five, and speed (cycle time) improvements up to a factor of two.

Due to their heuristic nature and limited scope, peephole optimizations can greatly benefit from automated verification. As unconventional designs, it is important that such optimizations be proven correct. Because of the small number of components involved, state explosion can be alleviated by localizing the verification.

On the other hand, flat verification of peephole optimizations against module specifications poses special problems. First, such verifications must usually take into account intricacies of switch-level and relative-time behaviors. Second, changes in the module interfaces need special techniques for modeling and incorporating the assumptions that justified the optimisations.

3. A STRUCTURED APPROACH FOR PEEPHOLE VERIFICATION

3.1 Assume-Guarantee Verification

Using hierarchical verification, a verification task p_1 ? $p_2 \underline{\ll} q$ can be split into three separate tasks that might have lower computational costs overall:

$$q_1 ? q_2 \underline{\ll} q; p_1 \underline{\ll} q_1; p_2 \underline{\ll} q_2. \qquad (3\text{-}1)$$

Here, p_1 and p_2 represent two submodules in the implementation, q_1 and q_2 represent the intermediate specifications of the respective submodules, and q represents the overall specification implemented by p_1 and p_2.

A frequently encountered difficulty, however, is loss of context information if operating assumptions are not modeled. Modules p_1 and p_2 may only “work correctly” in the presence of each other, in which case the refinement relations $p_1 \underline{\ll} q_1$ and $p_2 \underline{\ll} q_2$ only hold under certain assumptions. In Fig. 2, the OR gate does not directly refine the XOR gate specification. Nevertheless, if the environment guarantees that the inputs are mutually exclusive, a refinement check in context holds, verifying that the OR gate safely implements the XOR; see Example 1 in Subsection 3.2.

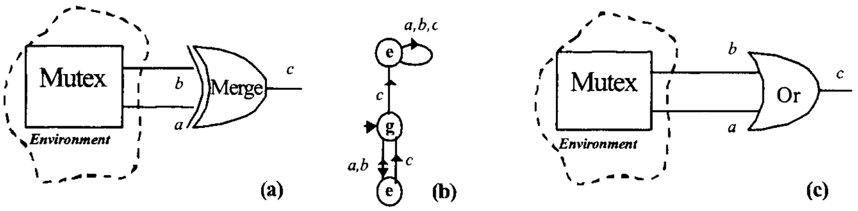

Figure 2. Replacement under assumptions: (a) XOR in a certain environment; (b) Inertial XOR; (c) OR in a certain environment.

In verification methods based on assume-guarantee rules (such as [5], [15], [10], and [11]), the correctness of the implementation relies on assumptions from the environment. According to assume-guarantee rules, a verification of the form p_1? $p_2 \underline{\ll} q$ can be decomposed as follows:

$$p_1 ? q_2 \underline{\ll} q ? q_1; \qquad (3\text{-}2)$$
$$p_2 ? q_1 \underline{\ll} q ? q_2. \qquad (3\text{-}3)$$

More precisely, it suffices to verify (3-2) and (3-3) for some q_1 and q_2 of a certain restricted form in order to establish p_1 ? $p_2 \underline{\ll} q$. However, note that q_1 and q_2 cannot be arbitrary. For instance, empty accessible sets of q_1 and q_2 trivially satisfy the decomposed verification tasks. Special conditions must be satisfied by q_1 and q_2 to break the circularity of reasoning and establish validity of p_1 ? $p_2 \lesssim q$. In previous methods, q_1 and q_2 are selected to have non-empty prefix-closed accessible sets, which justify a structural induction argument. In our method, we extend the rule so that q_1 and q_2 are arbitrary.

3.2 The Peephole Rule

Suppose that, in the original verification task, the refinement relation

$$p_1? \ \ldots r_i \ ? \ \ldots \ ? \ p_n \lesssim q$$

is checked. Here p_i (i=1..n) and q are arbitrary processes, representing the implementation components and specification, respectively; r_i is an arbitrary process, call it *peephole* replacement, which replaces $p_i$in a peephole optimization. Now, let d be an arbitrary process, call it *optimization assumption,* which formalizes the designer's hypothesis that made the replacement possible; and, let M be an arbitrary set of processes, call it *support model,* representing modules in the closed system, consisting of the implementation and the environment, that will guarantee the validity of the optimization assumption.

Theorem 1. *For processes p_1 , ..., p_n,q, r_i, d and process set M:*

if $p_1? \ \ldots p_i \ ? \ \ldots ? \ p_n \lesssim q$
and $(? \ d \ ? \ S_E, \ M \ ? \ \{p_j \,|\, j?i\} \ll \{-q\}$:
$r_i \ ? \ d \lesssim p_i \ ? \ (?_{m?M} \ m) \lesssim (?_{m?M} \ m) \ ? \ d \ )$
then $p_1 \ ? \ \ldots r_i \ ? \ \ldots \ ? \ p_n \lesssim q$.

We can phrase Theorem 1 informally as follows: if r_i refines p_i under constraint d, and d imposes no additional confinement over the system, then r_i can replace p_i in a refinement relation.

Proof:

$r_i \ ? \ d \lesssim p_i$
? ? (by monotonicity w.r.t. product by $(?_{j?i} \, p_j) \ ? \ (-q)$)
$(?_{j?i} \, p_j) \ ? \ (-q) \ ? \ r_i \ ? \ d \lesssim (?_{j?i} \, p_j) \ ? \ (-q) \ ? \ p_i$
? ? (by commutativity of product)
$(?_{j?i} \, p_j) \ ? \ (-q) \ ? \ r_i \ ? \ d \lesssim (?_k p_k) \ ? \ (-q)$ (1)

$(?_{m?M} \ m) \lesssim (?_{m?M} \ m) \ ? \ d$ and $M \ ? \ \{p_j \,|\, j?i\} \ll \{-q\}$
? (by montonicity of product w.r.t. components from outside M)
$(?_{j?i} \, p_j) \ ? \ (-q) \lesssim (?_{j?i} \, p_j) \ ? \ (-q) \ ? \ d$
? (by montonicity of product w.r.t. r_i)
$(?_{j?i} \, p_j) \ ? \ (-q) \ ? \ r_i \lesssim (?_{j?i} \, p_j) \ ? \ (-q) \ ? \ d \ ? \ r_i$ (2)

(1) and (2)
? (by transitivity, since $(?_k p_k) \lesssim q$? $(?_k p_k)$? $(-q)$? $p_i \lesssim$?)
$(?_{j?i} p_j)$? $(-q)$? $r_i \lesssim$?
?
p_1? ... r_i ? ... ? $p_n \lesssim q$

Since Theorem 1 is specifically developed for verifying peephole optimizations, we refer to Theorem 1 as the *peephole rule.*

Example 1: The example in Fig. 2 illustrates a peephole optimization that replaces an XOR gate by an OR gate in a mutual exclusion environment. In this example, the peephole implementation is an inertial XOR gate; the peephole replacement is an inertial OR gate; the optimization assumption is "Pulses on *a* and *b* never overlap"; and, the support model is a Mutex component existing in the system.

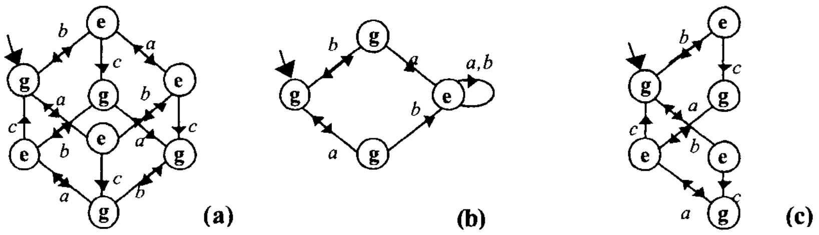

Figure 3. Usage of the peephole rule: (a) Inertial OR gate in open system; (b) Context constraint; (c) Process of OR gate under environment assumptions (context constraint). (Omitted edges on *a, b, c* lead to a permanent escape state.)

In the verification of Fig.2, assume refinement relation

$$p_{\mathrm{XOR}} ? \cdots p_{mutex} ? \cdots \lesssim q$$

holds, in which p_{XOR} represents the XOR gate and p_{mutex} represents an environment component which physically enforces mutual exclusion of signals *a* and *b.*

Let $M = \{p_{mutex}\}$, and let d be mutual exclusion constraint in Fig. 3(b). Note that d is the product of the following relative timing constraints:

$D(a{+}\, a{-}) < D(a + b{+})$;
$D(b{+}\, b{-}) < D(b{+}\ a{+})$.

Because

$$p_{mutex} \preceq p_{mutex} ? d$$

and

$$p_{\mathrm{OR}} ? d \preceq p_{\mathrm{XOR}},$$

using the peephole rule, we have:

$$p_{\mathrm{OR}} ? \ldots p_{mutex} ? \ldots \preceq q.$$

Thus, the OR gate can safely replace the XOR gate in this system. ?

Theorem 1 relates to the assume-guarantee rule over a particular case. If p_1 from (3-2) is substituted by r_i from Theorem 1, q_1 from (3-2) is substituted by p_i from Theorem 1, and q_2 from (3-2) is substituted by the process $(?_{j?i}\, p_j)$, then the hypothesis of Theorem 1 implies inequation (3-2), which is part of the hypothesis of the assume-guarantee rule.

$$r_i ? d \preceq p_i \ ? \ r_i ? (?_{j?i}\, p_j) ? d \preceq p_i ? (?_{j?i}\, p_j)$$
$$r_i ? (?_{j?i}\, p_j) \preceq p_i ? (?_{j?i}\, p_j) = p_i ? (?_{j?i}\, p_j) ? p_i \preceq q ? p_i$$

The remaining part of the hypothesis of the assume-guarantee rule, inequation (3-3), also follows from the hypothesis of Theorem 1 under the substitutions above, while p_2 from (3-2) can be substituted by any process p that refines $(?_{j?i}\, p_j)$.

Theorem 1 also relates to hierarchical verification over a particular case. If p_j is the intermediate specification for r_i, and M is the empty set, then $(?_{m?M}\, m) =$? . Since ? is the most transparent process in a process space (Definition 2.13 in [17]) then $d =$? . In this case, repeated application of Theorem 1 matches the hierarchical verification procedure as described by inequation (3-1).

3.3 Heuristics for Finding Verification Assumptions

Not all the optimization assumptions are guaranteed by the environment of the peephole alone. Since Theorem 1 has no restrictions on the connectivity of the processes involved, an optimization assumption may overlap both the peephole and the peephole environment. If the optimization assumption overlaps the peephole, we call the respective optimization assumption a *design assumption;* otherwise, we call the respective optimization assumption an *environment assumption.* Design assumptions are derived from properties of the circuit under verification which are known to the designer or verifier.

Notice that the peephole rule has no restrictions on the choice of the support model or the optimization constraint: system process subset M and process d in Theorem 1 are arbitrary. On the other hand, a poor selection of the support model will not lead to reductions of computational costs. For proper selection of the support model, one should consider the context of the circuit under verification. For example, in Fig. 3(c), the process of “OR gate under environment assumption” has fewer accessible executions than the process in Fig. 3(a) “OR gate in open system”, because some of the possible behaviors are eliminated by an optimization assumption based on a support model which involves not just circuit elements, but also the environment of the circuit.

In our experiments, the costs of verification based on peephole rules are influenced mainly by a tradeoff between the complexity and the determinism of the assumptions used.

?? Keep the assumptions simple, as the complexity of verification increases with the number of assumption processes.

?? Make the assumptions efficient by eliminating as many as possible of the “don’t-care behaviors” of the circuit under verification.

For instance, in the OR gate example above, a more effective assumption would only allow the pulses of *a* and *b* to alternate with pulses on *c*. If such an assumption can be guaranteed by the environment of the peephole, then not only we can perform stronger optimizations, but also we can verify them with less costs.

Presently, we mostly use relative timing constraints as optimization assumptions. By using relative timing constraints, verification does not need to start from a complete environment model, because a few hints from an incomplete environment model may suffice to guarantee the respective delay constraint as optimization assumption.

4. VERIFICATION OF THE ASP* ARBITER

A high-speed arbiter using the asP* protocol is reported in [8], with a non-optimized version and a speed-optimized version. The non-optimized implementation has good modularity, in the sense that the simple interfaces of the submodules achieve decoupling of the submodule designs. The optimized implementation achieves higher speed at some costs in modularity, by including more signals in the submodule interfaces. The high-level verification of the non-optimized version of this arbiter was reported in [13]. In this paper, we focus on the verification of the peephole optimisations.

4.1 Verification Strategy for the asP* Arbiter

The block diagram of the non-optimized version of the arbiter from [8] is shown in Fig. 4(a). The arbiter receives request events as input pulses and issues grant events as output pulses, after arbitration. Signals q_1 and q_2 are initially high, and the other signals are initially low. The rlatch component is a positive edge-triggered SR latch. When it receives input pulses (r_1 or g_1 for rlatch1), the rlatch converts them to output signal levels (y_1 for rlatch1). The dlatch in Fig. 4(a) converts pulses on the inputs to levels on the output. The two NOR gates generate grant pulses. The Mutex component ensures mutual exclusion between requests, and it is a four-phase component. For example, when request pulses from two channels arrive, rlatch1 and rlatch2 set y_1 and y_2 high. The Mutex arbitrates the input and gives grant to one, e.g. channel1, then q_1 is set to low and fires the rising edge of g_1. The rising edge of g_1 is propagated to dlatch and set f high, thereby resetting g_1; at the same time, feedback of g_1 pulse withdraws the request y_1. The pending request from channel2 gets a grant from Mutex and grant pulse g_2 to channel2 will be issued after an acknowledge pulse d_1 from channel1 is received. Notice that the grant pulse will be issued without waiting for the falling edge on the request pulse; only the rising edges of the *r*, g, and *d* signals are active.

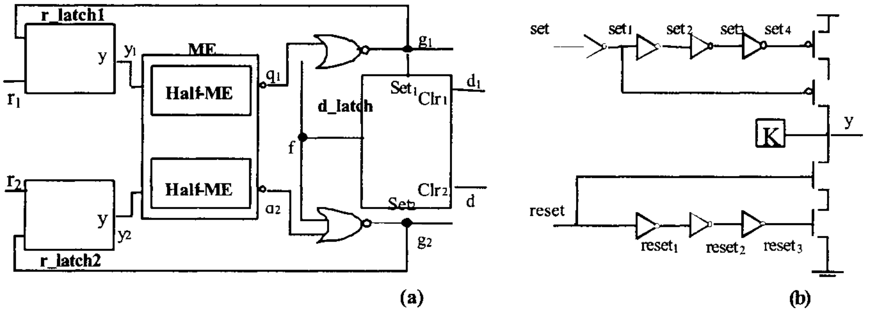

Figure 4. asP* arbiter before optimisations: (a) High-level implementation; (b) Low-level implementation: rlatch

To prove correctness of this arbiter, we aim to establish refinement between the high-level specification and the switch-level implementation. We apply the hierarchical verification procedure described in Section 3, using the submodules in Fig. 4(a) as intermediate specifications. In the following, $p_{Mutex}, p_{rlatch1}, p_{rlatch2}, p_{dlatch}, p_{\mathrm{NOR1}}$ and p_{NOR2} denote processes for the switch-level implementation of submodules, q_{Mutex}, $q_{rlatch1}$, $q_{rlatch2}$, q_{dlatch}, q_{NOR1} and q_{NOR2} denote processes for the submodule interfaces, and $q_{arbiter}$ denotes the process for the overall transition-event specification of the arbiter. The verification proceeds on two levels, as follows:

On the first level, the refinement between the connection of intermediate specifications and overall specification is checked:

$$q_{Mutex} \; ? \; q_{rlatch1} \; ? \; q_{rlatch2} \; ? \; q_{dlatch} \; ? \; q_{\mathrm{NOR1}} \; ? \; q_{\mathrm{NOR2}} \lesssim q_{arbiter}$$

Next, the refinements of submodule specifications against their switch level implementations are checked under certain environment assumptions:

$$p_{Mutex} \underline{\ll} q_{Mutex};\quad p_{rlatch1} \underline{\ll} q_{rlatch1};\quad p_{rlatch2} \underline{\ll} q_{rlatch2};$$
$$p_{dlatch} \underline{\ll} q_{dlatch};\quad p_{\mathrm{NOR1}} \underline{\ll} q_{\mathrm{NOR1}};\quad p_{\mathrm{NOR2}} \underline{\ll} q_{\mathrm{NOR2}}.$$

In this paper, we call the verification on the first level high-level verification and the second level submodule verification. For the details of high-level verification and overall specification, we refer readers to [13].

4.2 Rlatch Verification

Submodule verification consists of refinement checks between switch level implementation of submodules and their intermediate specifications. We use rlatch as the example of refinement checking on this level. The representations of implementations at switch level follow [17].

The implementation of the rlatch from [8] is shown in the Fig. 4 (b). When a rising edge of signal *set* comes, the inverter chain of set (contains 4 inverters) generates a pulse upon the PMOS network, thus setting output *y* high. The same mechanism is used for reset. The box labeled *K* is a “keeper” circuit consisting of a weak inverter and a feedback inverter that form a loop, so that the value of the rlatch output can be kept when there are no set or reset paths enabled.

The rlatch is verified by checking the following refinement relationships:

$$p_{rlatch} = p_{set\_inv\_chain}\ ?\ p_{reset\_inv\_chain}\ ?\ p_{\mathrm{MOS}} \underline{\ll} q_{rlatch}$$

in which $p_{set\_inv\_chain}$ is the model of set inverter chain (4 inverters); $p_{reset\_inv\_chain}$ is the model of reset inverter chain (3 inverters); p_{MOS} is the model of the reset part of rlatch.

Some assumptions can be drawn from the environment of the rlatch in the asP* arbiter system. Interface signals *set*, *reset* and *y* of the rlatch are under constraints. For instance, in the channel1 of asP* arbiter, grant signal g_1 is used as reset signal of rlatch1. Notice that for rlatch1, g_1+ will not be triggered until y_1+ is propagated through Mutex and NOR gate. As a result, we can make an environment assumption for rlatch:

$$d_{env}\text{:}\ \ D\ (set{+}\ reset{+}) > D\ (set{+}\ y{+})$$

Correspondingly, the verification of the rlatch becomes:

$$d_{env}\ ?\ p_{rlatch} \underline{\ll} q_{rlatch}$$

The verification result shows that the refinement relationship holds when the input pulse is wide enough to be caught by the inverter chain. This relative timing constraint is easily implemented by sizing, thus the rlatch implementation in Fig. 4(b) refines rlatch specification in typical environments for the asP* arbiter.

4.3 Peephole Optimizations of the asP* Arbiter

Fig. 5 (a) is the block diagram of asP* arbiter after optimization in [8]. Notice the change of submodules at their interfaces. (For example, the new interface of Mutex has 12 signals.) In Fig. 5 (a), signal *rr* of the Half_Mutex (Mutex/2 blocks in Fig. 5(a)) is connected to the internal nodes set_2 of rlatch (see fig. 4(b)); signal g_b is connected to the internal node set_1 of dlatch, which has the same setting-resetting structure as that for the rlatch [8].

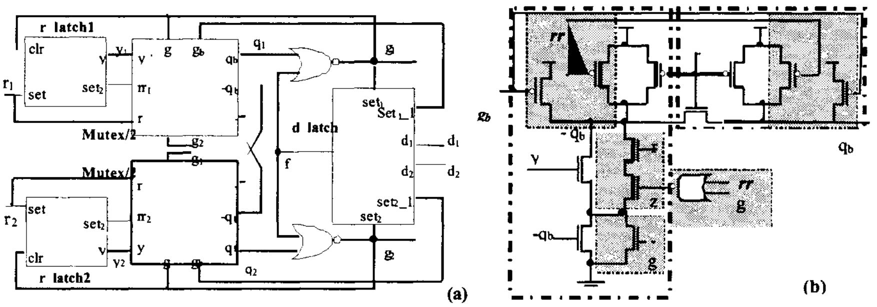

Figure 5. Peephole Optimization of asP* arbiter: (a) asP* arbiter block diagram after optimization; (b) Half-Mutex after optimisation.

Fig. 5 (b) shows the peephole optimizations over the half Mutex. Boxes in dark indicate modifications during the optimizations. Signals *rr* and g_b from the internal nodes of rlatch and dlatch are highlighted. For the details and motivation of these optimizations, we refer the reader to [8].

Notice that only Mutex's low-level structure is changed by optimization. For the other submodules, optimization only changes the interfaces by making some internal signals visible. As shown in Fig. 6(a), modules *rr*1, *rr*2, *gb*1 and *gb*2 are "abstracted" from rlatch1, rlatch2 and dlatch. Relations between models hold as follows:

$$p_{rlatch1} = p_{rr1} ? p_{rlatch1};\ p_{rlatch2} = p_{rr2} ? p_{rlatch2};\ p_{dlatch} = p_{gb1} ? p_{gb2} ? p_{dlatch}$$

By this module partition, we isolate unchanged modules (white blocks) in Fig. 4(a) from the models changed by optimization (dark blocks). The verification task after peephole optimization verification is:

$$p_{gb1} ? p_{gb2} ? p_{rr1} ? p_{rr2} ? p_{Mutex'} ? q_{rlatch1} ? q_{rlatch2} ? q_{dlatch} ? q_{\mathrm{NOR1}} ? q_{\mathrm{NOR2}} \underline{\ll} q_{arbiter}$$

In which $p_{Mutex'}$ represents the replacement of p_{Mutex} in optimization.

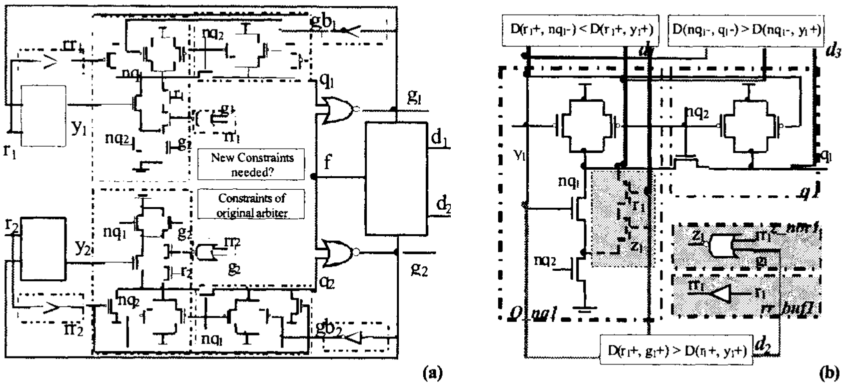
New Constraints needed?
Constraints of original arbiter
D(r₁+, nq₁-) < D(r₁+, y₁+)
D(nq₁-, q₁-) > D(nq₁-, y₁+)
D(r₁+, g₁+) > D(r₁+, y₁+)
(a)
(b)

Figure 6. Peephole Optimization verification of asP* arbiter: (a) asP* arbiter after optimization; (b) Verification and result of optimizationl (half).

4.4 Verification of Peephole Optimizations

Applying the peephole rule, the verification of the peephole optimised Mutex amounts to verifying the following relation, for some constraint *d*:

$p_{Mutex'}\ ?\ d \lll q_{Mutex}$

and to verifying that constraint *d* does not impose undue confinement on the system, by finding a suitable support model for *d*.

There are three optimizations addressed in [8]. The first optimization over half-Mutex is shown in Fig. 6(b). Changes involved in this optimization are highlighted. To apply the peephole rule, we take the following preparation steps:

?? First, a model of the optimization assumption is constructed. In Fig. 6(b), the highlighted part indicates the optimization assumption of optimizationl, which "adds an additional 'bypass' to allow the rising edge of a request to be applied directly to the arbiter without incurring the delay of the rlatch" [8]. The relative timing constraint d_1: $D(r_i+ nq_i-) > D(r_i+y_i+)(i = 1, 2)$ in Fig. 6(b) comes from this statement.

?? Second, the refinement $p_{gb1}\ ?\ p_{gb2}\ ?\ p_{rr1}\ ?\ p_{rr2}\ ?\ q_{Mutex}\ ?\ q_{rlatch1}\ ?\ q_{rlatch2}\ ?\ q_{dlatch}\ ?\ q_{NOR1}$ $??\ q_{NOR2}\ ?\ d_1 \lll q_{arbiter}$ is checked and the relation holds.

?? In the last step, we choose $M = \{p_{gb1}, p_{gb2}, p_{rr1}, p_{rr2}, d_1\}$ as the initial support model of peephole verification, and optimization assumption $d = d_1\ ?\ p_{rr1}?\ p_{rr2}$.

Two constraints are introduced in the procedure. One constraint:

$$d_2: \quad D(r_i{+}\, g_i{-}) > D(r_i{+}\, y_i{+}) \quad (i = 1, 2)$$

has as support model the components $p_{rlatch\,1}$ and $p_{rlatch\,2}$. The result of peephole verification is:

$$M = \{\, p_{gb1}, p_{gb2}, p_{rr1}, p_{rr2}, d_1, q_{rlatch1}, q_{rlatch2});$$
$$d = d_1 \,?\, d_{rr1} ?\; d_{rr2} ?\; d_2;$$
$$p_{Mutex'} \,?\, d \,?\, d_3 \underline{\ll} q_{Mutex}$$

in which d_3 i s an extra delay constraint:

$$d_3: \quad D(nq_i{-}\, q_i{-}) > D(nq_i{-}\, q_i{+})(i = 1, 2)$$

needed for the holding of the refinement.

The second and third optimizations are verified in the same way and the refinement only holds under certain additional constraints.

To examine the validity of the additional constraints detected in the peephole verification, we changed peephole in verification from Mutex to the whole arbiter. Applying peephole rules, we verify the implementation in Fig. 6 (a). The verification terminated and the result reported that no other relative timing constraints are needed for arbiter optimization:

$$M = \{d_{\mathrm{des1}}, d_{\mathrm{des2}}\};\; d = d_{\mathrm{des1}} \,?\, d_{\mathrm{des2}}$$
$$p_{Mutex'} \,?\, q_{rlatch1} \,?\, q_{rlatch2} \,?\, q_{dlatch} \,?\, q_{\mathrm{NOR1}} \,?\, q_{\mathrm{NOR2}} \,?\, d_h \,?\, d \underline{\ll} q_{\,arbiter}$$
$$d_{\mathrm{des1}}: \quad D(r_i{+}\, nq_i{-}) < D(r_i{+}\, y_i{+}) \quad \text{from optimization 1 } (i = 1, 2)$$
$$d_{\mathrm{des2}}: \quad D(g_i{+}\, gb_i{-}) > D(g_i{+} y_i{+}) \quad \text{from optimization 2 } (i = 1, 2)$$

We conclude that the optimized arbiter in Fig. 5 (a) can substitute the arbiter in Fig. 4(a).

Acknowledgments

We thank Mark Greenstreet for numerous comments, and Mark Josephs for discussions that inspired Example 1. We gratefully acknowledge financial support from NSERC, FCAR, CFI, MEQ, and McGill University.

REFERENCES

[1] V. Akella and G. Gopalakrishnan. “SHILPA: A high-level synthesis system for self-timed circuits.” In *Int. Conf: Computer-Aided Design (ICCAD)*, pp. 587–591, 1992.

[2] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. “MOCHA: modularity in model checking.” In *Computer-Aided Verification (CAV)*, pp. 521-525, 1998.

[3] E. Brunvand and R. F. Sproull. "Translating concurrent programs into delay-insensitive circuits." In *Int. Conf. Comput. Design,* pp. 262–265,1989.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. "Model checking and abstraction." In *Proc. Symp. on Principles of Programming Languages,* pp. 343-354, 1992.

[5] E. M. Clarke, D. E. Long, and K. L. McMillan. "Compositional Model Checking." In *Proc. Annual Symp. on Logic in Computer Science (LICS),* pp. 353-362,1989.

[6] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT press, 1989. (An ACM Distinguished Dissertation.)

[7] G. Gopalakrishnan. "Peephole Optimization of Asynchronous Macromodule networks." *IEEE Transitions on Very Large Scale Integration Systems,* (7) 1: 1999.

[8] M.R. Greenstreet and T. Ono-Tesfaye. "A fast, asP* RGD arbiter." In *Proc. Int. Symp. on Advanced Research on Asynchronous Circuits and Systems,* pp. 173-85, 1999.

[9] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[10] T. Henzinger, S. Qadeer, and S. Rajamani. "You assume, We Guarantee: Methodology and Case Studies." In *Computer-aided Verification (CAV),* pp. 440-451, 1998.

[11] T. Henzinger, S. Qadeer, S. Rajamani, and S. Tasiran. "An Assume-Guarantee Rule for Checking Simulation." In *Proc. Int. Conf. on Formal Methods in Computer-aided Design (FMCAD),* pp. 421-432, 1998.

[12] S. Hauck. "Asynchronous design methodologies: an overview." *Proceedings of the IEEE,* Vol. 83, pp: 69-93, 1995.

[13] X. Kong, R. Negulescu. "Formal Verification of Pulse-Mode Asynchronous Circuits." In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC), Special Session on Asynchronous Circuits,* pp. 347–352, 2001.

[14] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. "Property preserving abstractions for the verification of concurrent systems." *Formal Methods in System Design,* (6): 1-35, 1995.

[15] K.L. McMillan. "A compositional rule for hardware design refinement." In *Computer-Aided Verification (CAV),* pp 24-35, 1997.

[16] C. E. Molnar, I. W. Jones, B. Coates, and J. Lexau. "A FIFO ring oscillator performance experiment." In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems,* 1997.

[17] R. Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits.* Ph.D. thesis, University of Waterloo, Canada, 1998.

[18] R. Negulescu. "Process spaces." In *Proc. Int. Conf. on Concurrency Theory (CONCUR),* pp. 196-210, 2000.

[19] R. Negulescu and A. Peeters. "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.

[20] J. Sifakis. "Property preserving homomorphisms of transition systems." In *Proc. 4th Workshop on Logics of Programs,* June 1983.