Formal Verification of Peephole Optimization in Asynchronous Circuits

Xiaohua Kong

Department of Electrical and Computer Engineering
McGill University, Montreal

March 2001

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the Degree of
Master of Engineering

Abstract

This thesis proposes and applies novel techniques for formal verification of peephole optimizations in asynchronous circuits. Our task is to verify whether locally optimized modules can replace parts of an existing circuit under certain assumptions regarding the operation of the optimized modules in context. Two main difficulties in verifying peephole optimizations are state explosion in the implementation models and increased complexity of the interfaces of optimized modules. A novel technique is proposed for constructing in a modular manner specifications and functional models of pulse-mode circuits. A verification rule related to assume-guarantee and hierarchical verification is presented, using relative timing constraints as optimization assumptions. We present two case studies to illustrate the proposed techniques: verification of speed-optimizations in an asynchronous arbiter, and verification of one step of communication refinement in a globally asynchronous, locally synchronous (GALS) architecture.

Résumé

Cette thèse propose et applique des techniques pour la vérification formelle des optimisations locales dans des circuits asynchrones. Notre tâche est de vérifier si les modules localement optimalisés peuvent remplacer des pièces d'un circuit existant dans certaines conditions concernant le fonctionnement des modules optimalisés dans le contexte. Deux difficultés principales en vérifiant des optimisations locales sont l'explosion du nombre des états dans les modèles de dispositif et la complexité accrue des interfaces des modules optimalisés. On propose une technique nouvelle pour construire de façon modulaire des modèles fonctionnels des circuits mode impulsion. On présente une règle de vérification liée aux règles "assume-guarantee" et à la vérification hiérarchique, en utilisant des contraintes relatives de synchronisation comme suppositions d'optimisation. Nous présentons deux études de cas pour illustrer les techniques proposées: vérification des optimisations dans un circuit arbitre asynchrone, et vérification d'une étape de raffinement de transmission dans une architecture globalement asynchrone et localement synchrone.

Acknowledgements

I would like to start by expressing my gratitude to Prof. Radu Negulescu for his continuous support, help, and patience. In fact, without his supervision and guidance, this work would have never converged.

I also like to thank my external examiner, Mark Greenstreet, for his insightful comments; Ian Jones, from Sun Microsystems Laboratories, who brought to my attention the asynchronous arbiter case study; my colleague in the Microelectronics and Computer Systems laboratory, Larry Ying, who worked together with myself on data communication refinement and provided the implementation models of the GALS wrapper in Chapter 6; and my friends, Weiwen Zhu, Mark De Clercq and Clarence Tam, who never ceased to show all sorts of support and encouragement.

My gratitude always goes to my closest friend, Muhua Li, for her arrival bringing love and care, and for her cooking skills preventing me from starvation through the critical days of the Master's program.

Finally, I would like to acknowledge the financial support for my Master's studies provided by McGill University and NSERC (Natural Sciences and Engineering Council of Canada).

Contents

ABSTRACT ..... **I**

RESUME ..... **II**

ACKNOWLEDGEMENTS ..... **III**

CONTENTS ..... **IV**

FIGURES ..... **VI**

CHAPTER 1. INTRODUCTION **7**

1.1. MOTIVATION 7
1.2. PREVIOUS WORK 8
1.3. ORGANIZATION OF THE THESIS 9

CHAPTER 2. PRELIMINARIES **10**

2.1. PROCESS SPACES 10
2.1.1. Basic definitions *10*
2.1.2. Models of CMOS Cells *11*
2.1.3. Safety and Finalization Processes *13*
2.1.4. Relative Timing Constraints *13*
2.1.5. Operations *15*
2.1.5.1. Composition 15
2.1.5.2. Robustness 16
2.1.5.3. Refinement 16
2.1.5.4. Hiding 17
2.1.5.5. Reflection 19
2.2. FIREMAPS 20
2.3. EXAMPLE: VERIFICATION OF A PULSE-MODE JOIN 21
2.4. MODULAR AND HIERARCHICAL VERIFICATION 22
2.5. PEEPHOLE OPTIMIZATIONS IN ASYNCHRONOUS CIRCUITS 23

CHAPTER 3. CONSTRUCTING SPECIFICATIONS FROM HIGH-LEVEL EVENTS 25

3.1. PULSE-MODE HANDSHAKING 26
3.2. PULSE EVENTS AND TRANSITION EVENTS 28
3.3. TRANSITION-EVENT SPECIFICATION CONSTRUCTION 29

CHAPTER 4. A STRUCTURED APPROACH FOR PEEPHOLE VERIFICATION **32**

4.1. ASSUME-GUARANTEE VERIFICATION 32
4.2. THE PEEPHOLE RULE 34
4.3. HEURISTICS FOR FINDING VERIFICATION ASSUMPTIONS 38

4.4. STRATEGY FOR ASSUME-GUARANTEE VERIFICATION 39
CHAPTER 5. CASE STUDY: A HIGH-SPEED ARBITER 41
5.1. VERIFICATION BEFORE OPTIMIZATIONS 41
5.1.1. High-level Verification 43
5.1.1.1. Construction of asP* Arbiter Specification 43
5.1.1.2. Implementation Description 45
5.1.1.3. Verification Result 45
5.1.2. Submodule Verification 49
5.1.2.1. rlatch implementation 49
5.1.2.2. Switch-level Verification 50
5.2. VERIFICATION OF THE PEEPHOLE-OPTIMIZED OF ASP* ARBITER 52
5.2.1. Peephole Optimizations of the asP Arbiter 53*
5.2.2. Verification of Peephole Optimizations 55
CHAPTER 6. CASE STUDY: COMMUNICATION REFINEMENT 59
6.1. DATA TRANSFER SPECIFICATION 59
6.1.1. Communication Features 59
6.1.2. Fusion Processes 62
6.2. GALS WRAPPER VERIFICATION 64
6.2.1. GALS Wrapper 64
6.2.2. High Level Verification Result 65
CHAPTER 7. CONCLUDING REMARKS 68
REFERENCES 69

Figures

Figure 2-1: Example processes: 11
Figure 2-2: Relative timing constraint: 14
Figure 2-3: Hiding an action in a process: 18
Figure 2-4: Dual-edge pulse-mode JOIN : 20
Figure 3-1: Single-rail data interface between sender and receiver 26
Figure 3-2: 2-phase handshaking protocol 27
Figure 3-3: asP* handshaking protocol 27
Figure 3-4: Dual-edge pulse mode handshaking protocol. 28
Figure 3-5: JOIN element and its specification: 29
Figure 3-6: Construction of the dual-edge pulse-mode JOIN specification: 30
Figure 4-1: Replacement under assumptions: 33
Figure 4-2: Usage of the peephole rule: 36
Figure 5-1: Block diagram of the asP* arbiter before optimizations. 41
Figure 5-2: Construction of the asP* arbiter specification: 43
Figure 5-3: Constructing intermediate specification of rlatch: 44
Figure 5-4: High-level verification result of the asP* arbiter 47
Figure 5-5: Witness execution analysis. 48
Figure 5-6: Implementation of rlatch and dlatch: 49
Figure 5-7: Node models : 50
Figure 5-8: Model of node with Keeper. 51
Figure 5-9: Peephole Optimization of asP* arbiter: 53
Figure 5-10: asP* arbiter after optimization. 54
Figure 5-11: Reducing verification complexity by using the peephole rule. 55
Figure 5-12: Verification and result of optimization1 (half). 56
Figure 6-1: General data communication diagram. 60
Figure 6-2: Propagation of validity events. 61
Figure 6-3: Examples of Fusion Processes: 62
Figure 6-4: Specification constructed from fusions. 63
Figure 6-5: Data channel between two independently clocked domains. 64

Chapter 1. Introduction

1.1. Motivation

The dramatic increase of integrated circuit sizes 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. As an alternative to simulation, formal verification is an approach that exhaustively checks the correctness of a system. In this thesis, 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*.

One of our case studies involves verification of peephole optimizations in a high-speed asP* arbiter design. There, we were confronted with two main difficulties: 1) how to construct the specification properly and efficiently, and 2) how to avoid state explosion in verification.

Correspondingly, we propose techniques to solve or alleviate the two difficulties above, including: 1) construction of specifications for pulse-mode circuits from simpler high-level models, and 2) adapting assume-guarantee and hierarchical verification rules to the verification of peephole optimizations. Several asynchronous designs from [GO99], [MVK+99] and [PU98] are studied by applying our techniques. These circuits were found

to be correct according to our verification criteria, but only after including additional relative timing constraints that were not fully documented in the respective papers. These additional constraints are expressed in the form of chain constraints from [Ne98]. Although these additional constraints suffice to guarantee correctness, they are not unique; other forms of relative timing constraints may achieve the same effect.

1.2. Previous work

The framework for formal verification used in this thesis is process spaces [Ne98] [Ne00], in which most of the theory basis is developed. This thesis will apply process spaces to peephole optimizations in general, and to the analysis of two previously published circuits in particular. Metric free relative timing constraints were proposed in [Ne97], [Ne98], [St99], [Cort00] for verification and synthesis; a novelty of our approach is to use such constraints as optimization assumptions to automatically verify peephole optimizations. Our pulse-mode specification construction technique turned out to be close to the *handshake expansion* operation used, for instance, in [KCK+99] for constructing specifications for synthesis, and it also relates to CSP-like channel specifications as in [Be93]; a difference is that we use the same formalism for both the simplified channel representation and the full specification, and we construct full specifications by existing parallel composition operation instead of specialized program translation rules.

Our hierarchical verification is related to the methods in [CLM89] and [Di89]. Assume-guarantee rules are addressed in [Sta85], [CLM89], [GL94], [AL95], [Mc97], [HQR98], [HQR+98]. In this thesis we adapt the assume-guarantee rules using relative timing constraints, and 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 computation costs, as explained in Chapter 5.

1.3. Organization of the Thesis and Contributions

The presentation of this work proceeds as follows: Chapter 2 presents an overview of the models used and describes the preliminaries of peephole optimization in asynchronous circuits. In Chapter 3, we present our procedure for constructing specifications from simplified higher-level models; this procedure is illustrated by pulse-mode circuits. In addition, we describe the implementation of these constructs in FIREMAPS. Chapter 4 addresses our peephole verification rules and their application by using relative timing constraints. Chapter 5 presents the verification of an asP* arbiter as one of the case studies in the application of two methods introduced in Chapter 3 and Chapter 4 in pulse-mode circuit verification. As a more general case, communication refinement is studied in Chapter 6, which is part of the co-work with Larry Ying. Finally, the conclusions of our work and the possible future steps are presented in Chapter 7.

The pulse-mode circuit verification, which includes part of Chapter 3 and high-level verification in Chapter 5, have been reported in [KN01a]; Chapter 4 and the verification of peephole optimizations in Chapter 5 have been reported in [KN01b]; part of the verification work in Chapter 6 has been reported in [KNY01].

The contributions reported in [KNY01] are partitioned between this thesis and [Yi01] as follows: Subsection 4.1 of [KNY01] (Implementation Model Construction) is claimed in [Yi01]; the overall data transfer verification part of Subsection 4.3 in [KNY01] is split equally between this thesis and [Yi01]; Sections 1, 2, and 5, and Subsection 4.2 of [KNY01] are general remarks and background information; the rest of the [KNY01] is claimed in this thesis.

Chapter 2. Preliminaries

2.1. Process Spaces

Here we briefly overview the necessary notions and properties of process spaces, following [Ne98] and [Ne00]; for more details, we refer the reader to the source references.

2.1.1. Basic definitions

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.

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. The process *void*, denoted by Φ, is $(E; E)$.

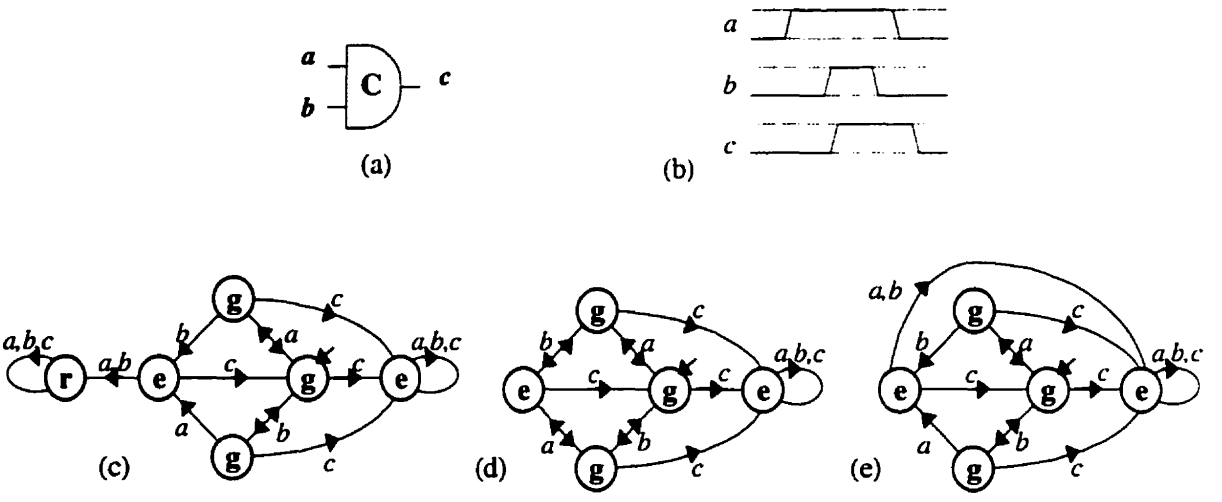


Figure 2-1: Example processes:
(a) C-element symbol; (b) Waveform;
(c) Hazard-intolerant model; (d) Inertial model; (e) Quickened model.

2.1.2. Models of CMOS Cells

Process spaces can be used to build models of circuit behavior in a manner similar to the state machines of Fig. 2-1. For an example of the models used in this thesis, consider the C-element in Fig. 2-1 (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 is unchanged. Waveforms are represented by finite sequences of actions corresponding to signal transitions, such as *abcbac* for the waveform in Fig. 2-1 (b). In this thesis, we use 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. 2-1 (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. 2-1 (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. Following [Ne98], by *hazard* we mean 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. 2-1. Hazard-intolerant models simply require the environment to avoid hazards, in which any execution that causes a hazard to the system is a reject execution.

The contract between the device and its environment can be adjusted by changing the model of the device. The model in Fig. 2-1 (d) is an *inertial* model. Informally speaking, a contract of inertial models implies that if the device believes there is a hazard from the environment, it can simply ignore the hazard. For example, in Fig. 2-1 (d), when input *a* is reset quickly and is considered as a hazard to the device, execution *a+ b+ b-* from initial state returns to the state which is reached by execution *a+* from initial state, as if the hazard on input *b* had never occurred. Without any assumptions from the environment, inverting CMOS cells can be modeled by inertial models, in which the input is considered as hazard when the hold time of an input pulse is shorter than the cell delay.

The model in Fig. 2-1 (e) is a *quickened* model: it assumes the device will take the responsibility to avoid hazards in the system by being quick to respond, instead of leaving the hazard-avoidance problem to environment as in the hazard-intolerant model. In the model in Fig.7 (e), the C-element should be "quick" enough so that no hazards will ever happen. Formally, the "slower" executions of device (such as *abb* for the C-element) are escapes because they should be avoided by a quick device (in our case, by issuing a *c* event before the second *b* event). In synchronous circuits, combinational logic between two registers can be modeled as "quick" enough with respect to clock events. After promising that the device is "quick enough" to simplify verifications, the designers should make good on that promise when choosing transistor sizes of the circuit later in the design flow. As will be shown later, a quickened model can be replaced by a hazard-

intolerant model plus a list of relative timing constraints, which can be implemented by sizing.

2.1.3. Safety and Finalization Processes

Informally speaking, *safety* properties represent that "something bad does not happen" [LL90]; in our applications, this amounts to avoidance of illegal events. *Finalization*, on the other hand, ensures that "something good does happen" (as it is both a liveness and a progress property); in our applications, this amounts to avoidance of illegal stopping. The safety and finalization properties can be expressed by means of sets of finite executions. In process spaces, safety and finalization are expressed as processes (pairs of acceptable and accessible sets of executions) over an execution set E equal to the set U^* of finite words where U is a universal set of actions.

Two different processes, one for safety and one for finalization, can be attached to the same system s. The *safety process*, denoted σs, deals with partial executions and records the occurrence of illegal inputs and outputs. The *finalization process*, denoted φs, is constructed for the total (complete) finite executions. In fact, the finalization process considers every sequence of actions a total execution and records the violations it contains. These violations include, in addition to illegal inputs and outputs, illegal stopping.

2.1.4. Relative Timing Constraints

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

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

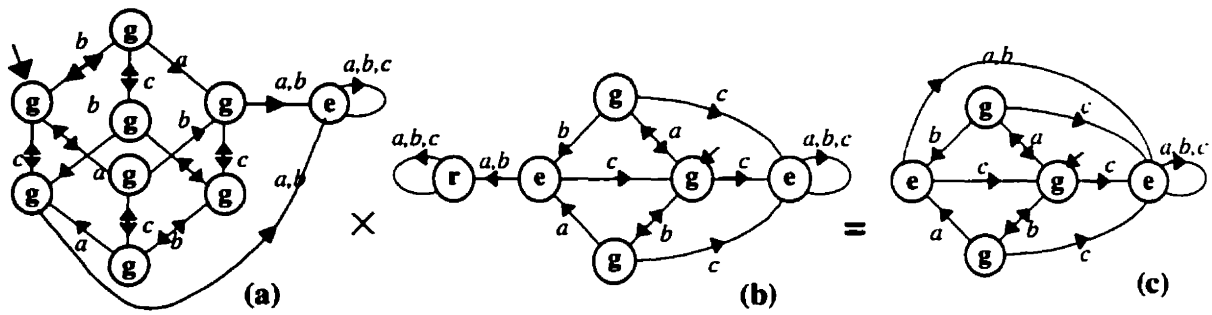


Figure 2-2: Relative timing constraint:
(a). Process of a chain constraint; (b) Hazard-intolerant model of C-element;
(c). Quickened model of C-element.

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* [NP98], imposes 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). For example, the "quicken model" in Fig. 2-1(e) implies the chain constraints:

$$D(a\,b\,b) > D(a\,b\,c); \qquad D(b\,a\,b) > D(b\,a\,c);$$
$$D(b\,a\,a) > D(b\,a\,c); \qquad D(b\,a\,b) > D(b\,a\,c);$$

Following [NP98], fig. 2-2 (a) shows the relative timing constraints represented in process automaton. With relative timing constraints in Fig. 2-2(a), the *HI* (Hazard-Intolerant) model of the C-element can be implemented to satisfy quickened assumption. Essentially, we obtain a chain constraint process by constructing a state machine that recognizes the chains and avoids executions that violate the relationship between the chain events. A device model under certain delay constraints can be constructed by the composition operation that we will discuss in section 2.1.5.1.

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. On the other hand, chain constraints can model the sizing assumptions of a device; to satisfy the specification, the implementation should follow these delay assumptions. Metric-free verification under

relative timing constraints was presented in [Ne98] and [Ne00]. Further, several verification and synthesis methods based on relative timing constraints have been introduced (see for instance [SGR99] and [PCKP00]).

We use *metric-free models* for relative timing constraints in verification. Only the relation between two chains is of concern, while how much delay occurs between two chains is not explicit. On the other hand, the method used here can model the delay that is distributed along a circuit path, without considering whether the delay is caused by components or wires.

2.1.5. Operations

2.1.5.1. Composition

Joint behavior (parallel composition) is expressed in process spaces by the *product* operation [Ne98]. The *product* of two processes p and q is a process $p \times q$ such that

$$\mathbf{as}(p \times q) = \mathbf{as}(p) \cap \mathbf{as}(q)$$
$$\mathbf{at}(p \times q) = (\mathbf{at}(p) \cap \mathbf{at}(q)) \cup \overline{(\mathbf{as}(p) \cap \mathbf{as}(q))},$$

or, equivalently, such that

$$\mathbf{r}(p \times q) = (\mathbf{r}(p) \cup \mathbf{r}(q)) \cap \overline{\mathbf{e}(p)} \cap \overline{\mathbf{e}(q)}$$
$$\mathbf{e}(p \times q) = \mathbf{e}(p) \cup \mathbf{e}(q)$$
$$\mathbf{g}(p \times q) = \mathbf{g}(p) \cap \mathbf{g}(q).$$

This means that the product of two processes can avoid some of the reject traces by the guarantee of executions from each other. Note that executions which are rejects to one factor process and escapes for the other factor process are escapes for the product process.

Notice that any two processes in a process space can be involved in the product operation, regardless of their connectivity.

Product is associative and commutative, and therefore can be applied to any number of processes. In addition, product extends to uncountable sets of processes.

2.1.5.2. Robustness

Robustness expresses the property of a process that the device it represents imposes no constraints on the environment it deals with. For safety processes, this means that the device accepts all the inputs it receives from the environment. Therefore, a process p is robust when it has an empty reject set $\mathbf{r}(p)$.

Consider, for example, the C-element models represented by the automata in Fig. 2-1. The hazard-intolerant model in Fig. 2-1 (c) is not a robust process, since the reject set is not empty. Execution *aba*, for instance, is a reject since it includes an illegal input (the second a transition cannot occur before the output changes). On the contrary, the process automata in Fig. 2-1 (d) and (e) represent robust processes that allow all inputs to happen at any state. (Inputs a and b are accepted in every state.) In in Fig. 2-7 (e), because it is the responsibility of the device to avoid the execution *aba*, the process is still considered as robust.

2.1.5.3. Refinement

Refinement is a relation between two processes that allows for one process p to fully replace a second process q. A process p is said to *refine* a process q, written $p \sqsupseteq q$, if

$$(\mathbf{at}(p) \supseteq \mathbf{at}(q)) \wedge (\mathbf{as}(p) \subseteq \mathbf{as}(q)),$$

or, equivalently, if

$$\mathbf{r}(p) \subseteq \mathbf{r}(q) \wedge \mathbf{e}(p) \supseteq \mathbf{e}(q).$$

Refinement is reflexive, transitive, and antisymmetric. In the case when refinement exists in both directions between two processes, i.e., $p \sqsupseteq q$ and $q \sqsupseteq p$, we say that the two processes are equal since they have equal accessible and acceptable sets.

Example.

To illustrate how to check the refinement relation between two processes, consider again the C-element models in Fig. 2-1. Name process models in Fig. 2-1 (c), (d), (e) *pc*, *pd*, and *pe* respectively. By comparing the states in the figures, one verifies that:

$$(\mathbf{e}(pd) = \mathbf{e}(pc)) \wedge (\mathbf{r}(pc) \supseteq \mathbf{r}(pd) = \varnothing)$$

Thus $pd \sqsupseteq pc$, or process *pd* can replace process *pc*.

Also, we have a refinement relation between (d) and (e):

$$(\mathbf{e}(pe) \supseteq \mathbf{e}(pd)) \wedge (\mathbf{r}(pe) = \mathbf{r}(pd) = \varnothing)$$

Thus $pe \sqsupseteq pd$, or process *pe* can replace process *pd*.

From the transitivity property of refinement, we have $pe \sqsupseteq pc$.

2.1.5.4. Hiding

As a particular case of the generic process abstractions in [Ne98] (Chapter 8), we define the hiding operation performed on processes. It follows from the general treatment in [Ne98] that hiding preserves refinement.

We start by defining the usual hiding operation for finite words. This operation eliminates occurrences of certain actions from a finite word.

Definition For alphabet U and a subset $B \subseteq U$, let $<\downarrow B> \subseteq U^* \times U^*$ be a binary relation on finite words over U such that:

a) $(\varepsilon,\varepsilon) \in <\downarrow B>$

b) $(u, v) \in <\downarrow B> \wedge a \in B \Rightarrow (ua, v) \in <\downarrow B>$

c) $(u, v) \in <\downarrow B> \wedge b \notin B \Rightarrow (ub, vb) \in <\downarrow B>$

d) each pair from $<\downarrow B>$ satisfies a), b), or c).

Let $<\downarrow B>'$ be the inverse of the relation $<\downarrow B>$.

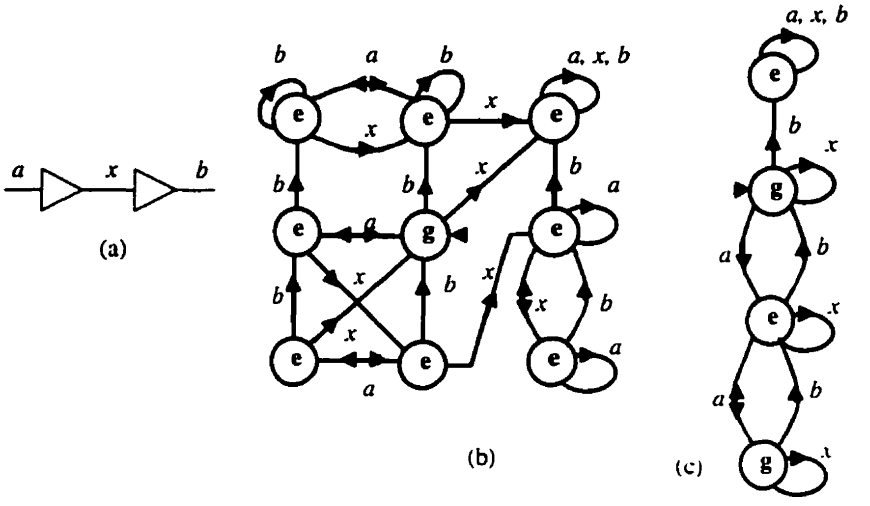


Figure 2-3: Hiding an action in a process:

(a) A system of two inertial buffers; (b) Product of the buffer processes:

(c) Result of hiding action x.

In words, $<\downarrow B>$ eliminates from an execution all actions from B, while $<\downarrow B>'$ inserts in an execution actions from B in arbitrary numbers and at arbitrary places. For example, we have the following:

$abcba <\downarrow\{b\}> aca$, $aca <\downarrow\{b\}>' abcba$, and $aca <\downarrow\{b\}>' bbacbbbab$.

The hiding operation on processes is then constructed using the generic construction for process abstractions in [Ne98].

For a subset X of U^*, let $<\downarrow B>X = \{<\downarrow B> u \mid u \in X\}$.

For a process p, let $\text{hide}_B(p)$ (read hiding of p over B) be defined such that:
$\mathbf{as}(\text{hide}_B(p)) = <\downarrow B>' <\downarrow B> \mathbf{as}(p)$, and
$\mathbf{r}(\text{hide}_B(p)) = <\downarrow B>' <\downarrow B> \mathbf{r}(p)$.

As an illustration, we use the example of the two buffers Fig. 2-3 (a), and their composition. The product process P needs not to observe the intermediate signal x. Therefore, we apply hiding to eliminate x from the alphabet of P. The resulting process of

product is shown in Fig. 2-3 (b) where the process still has action x. (The construction in Fig. 2-3 (b) was obtained by taking the Cartesian product of the state sets of the two buffer processes; the state machine can be simplified by collapsing the permanent escape states.) Subsequently, we eliminate action x from Fig. 2-3 (b) and we apply a standard determinization procedure. The result is shown in Fig. 2-3 (c), and it represents the behavior of a 2-bounded buffer.

Note that hiding followed by converse-hiding may produce more reject and accessible executions than the original process had. This amounts to saying that hiding and converse-hiding produce a pessimistic (conservative) approximation of a process by a hiding-independent process [Ne98].

2.1.5.5. Reflection

The *reflection* of a process p is a process $q = -p$ represented by

$\mathbf{as}(q) = \mathbf{at}(p)$ and

$\mathbf{at}(q) = \mathbf{as}(p)$.

Again, we can define reflection in terms of **r**, **g**, and **e**:

$\mathbf{r}(q) = \mathbf{e}(p)$

$\mathbf{e}(q) = \mathbf{r}(p)$, and

$\mathbf{g}(q) = \mathbf{g}(p)$.

Reflection represents an exchange of roles between device and environment.

Reflection reveals a link between robustness and refinement. For processes p and q:

$$p \sqsupseteq q \quad \Leftrightarrow \quad p \times (-q) \text{ is robust} \quad \Leftrightarrow \quad p \times (-q) \sqsupseteq \Phi,$$

where Φ, called *void*, is the identity element of product. Process Φ has only goal executions; its reject and escape sets are empty.

Thus, one can detect violations of refinement as counter-examples to the robustness of $p \times (-q)$, i.e., as executions that are rejects for $p \times (-q)$. Such counter-example executions

can be regarded as 'witnesses to failure' in the verification. Any execution that leads to failure is called a *witness execution.*

2.2. FIREMAPS

FIREMAPS was developed by R. Negulescu at the University of Waterloo. The name stands for **fi**nitary and **re**gular **ma**nipulation of **p**rocesses and **s**ystems. This tool automates the operations and manipulation of processes whose execution sets are regular languages of finite words.

All operations on processes discussed above are implemented in FIREMAPS. Composition, reflection, and hiding are examples of these operations. In addition, if robustness or refinement fail, FIREMAPS can produce a witness execution that provides a counter-example to the set relationships involved. Such witness executions serve for diagnosis by pinpointing faults.

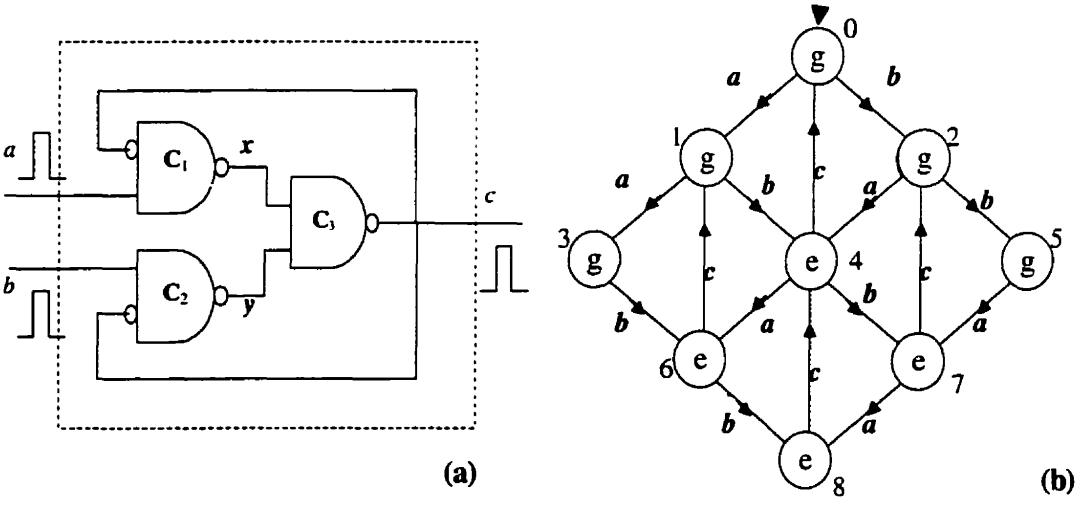


Figure 2-4: Dual-edge pulse-mode JOIN :
(a) Implementation; (b) Specification.

2.3. Example: Verification of a Pulse-mode JOIN

As an example of the verification method and gate models used, consider the specification in Fig 2-4 (b) and the implementation in Fig.2-4 (b), representing a pulse-mode JOIN element from [PU98]. The pulse-mode JOIN behaves like a C-element, except that: 1) it does not allow retraction of input events, and 2) the pulse-mode convention uses pulses rather than signal transitions to represent events. Thus, a pulse on output signal *c* is issued after pulses on input signals *a* and *b*. The implementation in Fig.2-4 (a) follows [PU98]. In Chapter 3, we discuss how we have arrived at the specification shown in Fig 2-4 (b), and we give more details about pulse-mode operation.

To avoid clutter in the figure, the specification in Fig 2-4 (b) does not indicate illegal transitions. We adopt the drawing convention that omitted illegal inputs lead to a permanent reject state and omitted illegal outputs lead to a permanent escape state. Also, omitted actions that are not visible at the interface of a module produce self-loops at every state of the corresponding process automaton.

The pulse-mode JOIN relies on timing constraints for correct operation. The result of our verification is that the refinement relationship does not hold without additional timing assumptions. A witness execution is as follows: *a+b+b-a-*. This witness execution is an escape for the specification, but a goal for the implementation. This witness execution represents a mismatch between the specification and implementation: two input pulses, after which the implementation stops without delivering an output pulse. This mismatch occurs if the C-elements used in the implementation are not quick enough to react to the input pulses.

On the other hand, we find that refinement holds if we use quickened models for the C-elements instead of inertial models. This change amounts to adding several relative timing assumptions, in the form of chain constraints that guarantee that the next input transition will give enough time for the C-elements to stabilize:

$$D\,(a+\ a-) > D\,(a+\ x-); D\,(b+\ b-) > D\,(b+\ y-); \ldots.$$

In other words, the implementation will work correctly if some minimal pulse widths are guaranteed. There is no need for upper bounds on the pulse widths.

2.4. Modular and Hierarchical Verification

Refinement checking can successfully find subtle errors in finite state systems, but it suffers from an inevitable problem, state explosion. Typical verification problems for multi-component systems are PSPACE-complete (see e.g. [Ne98]).

Several *structured verification* approaches can be used to alleviate state explosion in refinement checking. We can break the verification of a large system into several sub-tasks, and establish refinement individually for each subtask. Then, we infer by algebraic properties of the operations involved that refinement holds for the large system as well.

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 behavior of the higher level is treated as specification and the lower level description is treated as implementation. The description of circuit 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 are simpler (involve fewer components) than the overall verification task.

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

$$p \sqsupseteq q \;\Rightarrow\; p \times r \sqsupseteq q \times r.$$

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.

Since arbitrary processes can be involved in product and refinement, hierarchical verification can also involve processes that are at different abstraction levels. For example, a mixed gate-level and switch-level representation is often useful in our applications, where peephole optimizations might be applied only in critical parts of a circuit, while other parts might use standard gates.

2.5. Peephole Optimizations in Asynchronous Circuits

Peephole optimization is an approach to obtaining global optimizations of an integrated circuit design by successive local optimizations. Specifically, 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. As a particular case, in which replacement happens between different abstraction levels, the implementation of a component in a circuit can be considered as peephole optimization as well.

Peephole optimizations are often performed after high-level synthesis with significant gains. For example, two main high-level synthesis flows in asynchronous design translate concurrent program-like descriptions to asynchronous circuit composed of macro modules or handshake components (see e.g. [Pe96], [HBP+93], and [Go99]). In [Go99], a peephole optimization step after synthesis by the methods in [AG92] and [BS98] 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 cross-checking by automated methods. It is important that such optimizations be proven correct, which means that the observable behavior of the circuit after optimization meets the specification of the original circuit.

Hierarchical verification is particularly well-suited for post-synthesis peephole optimizations in asynchronous designs. First, as a result of high-level synthesis, the

macromodule networks usually have good modularity (although some modularity may be lost by altering the module interfaces for optimization purposes). Second, models for the behaviors of macromodules in the network are usually readily available.

On the other hand, the 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 optimizations so that the overall circuit specifications are met.

Chapter 3. Constructing Specifications from High-level Events

Specifications are usually assigned different interpretations at different stages in the design flow. Early in the design flow, specification is usually abstract and minimal, which is useful for analysis and reasoning by humans. When design details are settled, a specification explicitly comparable to the implementation is needed. In this sense, a specfication that only specifies high-level events would be inadequate for low-level verification. For example, in a case study of a arbiter verification, specification with high-level events only has 12 states, while the complete low-level specification has 768 states.

In order to bridge the gap between high-level and low-level specifications, a construction method of complete low-level specifications from smaller high-level specifications is introduced in this chapter. First, a specification using high-level events is represented as a process. Then, constraints that enforce correct occurrence of low-level transition events are modeled as processes. Finally, the low-level specification is built by computing a product of constraint processes and the high-level specification process.

3.1. Pulse-mode Handshaking

Unlike synchronous systems, in which a single control signal clock is used to synchronize the behaviors of each module, several asynchronous design styles use handshaking protocols to guarantee the correct data transfer between modules. The data transfer and its handshake control signals are shown in Fig. 3-1.

In the 2-phase handshaking protocol, a request and an acknowledge wire are used to implement the handshaking between the active and passive partners. Assuming that both signals start low initially, the sequence of events that can be observed on such a channel is depicted in Fig. 3-2. Although rising and falling transitions on the handshake control signals can be distinguished in other handshake protocols, they represent the same meaning in 2-phase handshaking.

Pulse-mode protocols, in which control signals are represented by pulses, are a family of protocols used in asynchronous circuit design. There are two types of pulse-mode protocols: *single-edge* and *dual-edge.*

The pulse mode protocol combines the advantage of the 2-phase protocol and 4-phase protocol. From the view of pulse event, the pulse mode protocol can work with the simplicity of 2-phase handshaking. At the same time, it starts the handshaking operation cycle always from the same state, thus keep the simplicity of implementation.

The single-edge mode, shown in Fig. 3-3, is named asP* in [MJCL97] and [GO99]. In the

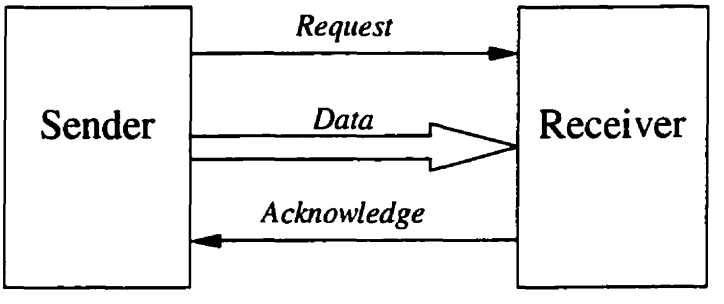


Figure 3-1: Single-rail data interface between sender and receiver.

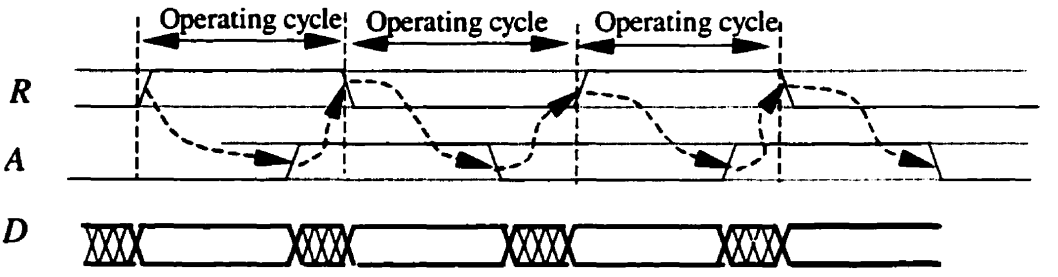


Figure 3-2: 2-phase handshaking protocol.

single-edge mode, only one edge of the pulse is used to transmit control information between modules, and the other edge is ignored. In the asP* protocol, the handshaking begins when the active edge of request (rising edge of *R* in Fig. 3-3) is issued, and it finishes when active edge of acknowledge (rising edge of *A* in Fig. 3-3) is received by the sender. The request and acknowledge signals may return to zero at arbitrary times.

Essentially, the asP* protocol is related to 4-phase handshaking (see e.g. [Pe96] for a description of 4-phase handshaking) by allowing more concurrency and thus more freedom in the implementation. In particular, the asP* protocol potentially achieves higher speeds, by removing the dependency of resetting, which simplifies the implementation. In our case study, we will use a high-speed arbiter from [GO99] that communicates with its environment via the asP* protocol.

In the dual-edge pulse mode, handshake events are represented by pulses rather than transitions on the control signals. Unlike asP* protocol, however, the falling edges of the

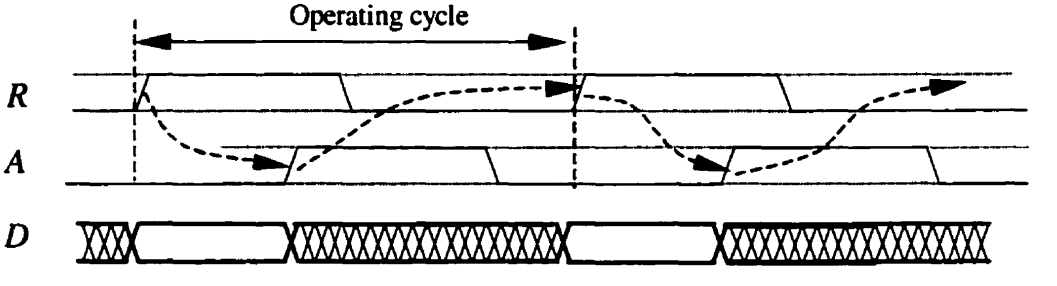


Figure 3-3: asP* handshaking protocol.

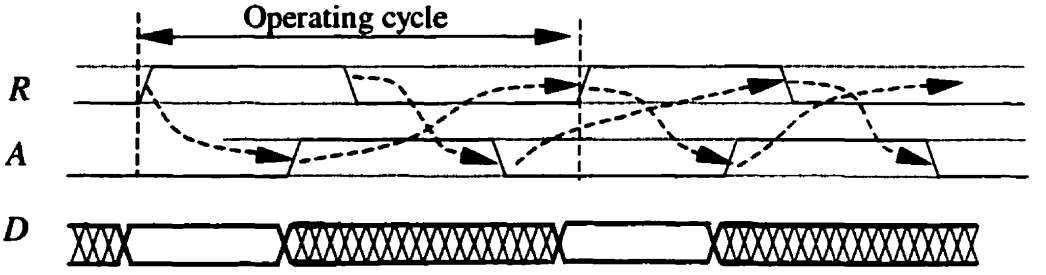


Figure 3-4: Dual-edge pulse mode handshaking protocol.

dual-edge pulse mode copy the sequence of the rising edges. The waveform of dual-edge handshaking protocol is shown in Fig. 3-4.

An example of a dual-edge pulse-mode component, JOIN element, was already introduced in Chapter 2, Fig. 2-4. The JOIN issues an output pulse only after it has received a pulse from each input. Notice that the falling edge of the output pulse can be produced only after the falling edges of the input pulses.

3.2. Pulse Events and Transition Events

The symbol and behavior of the pulse-mode JOIN element are defined as in Fig. 3-5 The actions in the automaton have an abstract meaning, and they represent high-level handshaking events. Each of the edges labeled a, b, or c stands for a pulse on the respective voltage signal. We call this kind of specification, in which each event represents a pulse, *pulse-event* specification. On the other hand, the behavior of the same JOIN element can also be described by events that represent individual signal transitions, as in Fig. 2-4 (b); we call such specifications *transition-event* specifications.

Although the arbiter in [GO99] was designed to a pulse-mode specification, its aggressively optimized implementation contains a mixture of pulse-mode, two-phase and level signaling. On the other hand, for comparison and verification it is necessary to assume that all processes give the same interpretation to executions. As transition-event

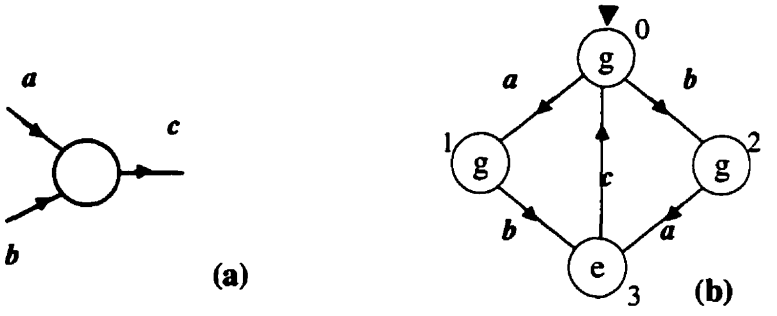


Figure 3-5: JOIN element and its specification:
(a) Symbol of JOIN element; (b) High-level (pulse-event) specification of pulse-mode JOIN.

specifications are more versatile than pulse-event specifications, we use transition-event specifications as the common denominator for verification purposes.

3.3. Transition-event Specification Construction

Building transition-event specifications by hand is tedious and error-prone for single-edge pulse-mode circuits, because of keeping track of many irrelevant interleavings of falling edges. For instance, the pulse-event specification of the RGD arbiter is a 12-state automaton, while its transition-event specification (with falling edges) has 768 states (after collapsing equivalent states).

To simplify the specification of pulse-mode and various mixed mode circuits, we decompose pulse-mode specifications into the following parts:

- For the single-edge mode, construct the pulse-event specification by describing only the *rising* edges of each pulse. This specification is similar to the transition-event specification of a two-phase circuit.

- For the dual-edge mode, construct two separate specifications, involving the *rising* edges only and *falling* edges only.
- Construct explicit constraints that the falling and rising edges of each signal should alternate.

These parts can be considered as distinct properties and we compose them to create a full transition-event specification.

In FIREMAPS, we use uniform models for all the specification parts mentioned above. We can construct full transition-event specifications by the following procedure, using process space operations:

- If single-edge, create automata that only consider rising edges (Fig. 3-6(a) is the rising edge automaton of pulse-mode JOIN). If dual-edge, create automata that consider only rising edges and separate automata that consider only falling edges (Fig. 3-6(b) is the falling edge automaton in dual-edge mode). These automata are

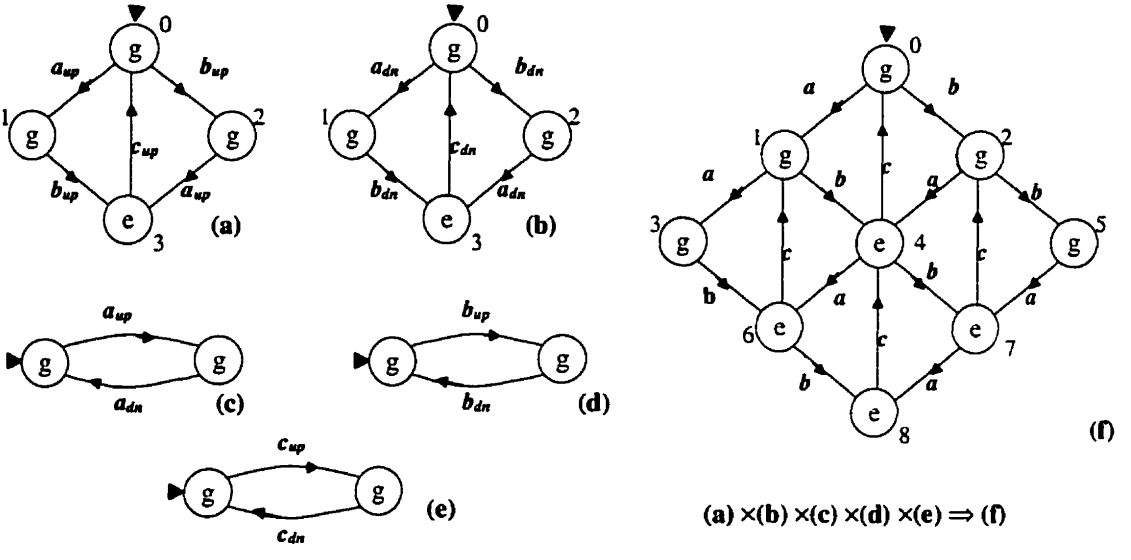


Figure 3-6: Construction of the dual-edge pulse-mode JOIN specification:
(a) Specification of rising edges; (b) Specification of falling edges;
(c)—(d) Pulse automata of signals a, b, c;
(f) Transition-event specification of dual-edge pulse-mode JOIN.

incomplete, in the sense that illegal events are disabled. For example, in Fig. 3-6(a), c_{up} (the rising transition on signal c) is disabled at state 0.

- Create automata that specify alternation of rising and falling edges of each pulse signal; call them *pulse automata*. In pulse automata, rising and falling edges have different symbols. (See Fig. 3-6(b), (c), and (d).) Pulse automata have a standard form and can be obtained by relabeling.
- Represent constraints between pulses (e.g. mutual exclusion), by automata.
- Compute the product of the process automata above. (In Fig. 3-6 produce (a), (c), (d) and (e) for single-edge mode specification and produce from (a) to (e) for dual-edge mode specification.) Missing edges in the component automata correspond to missing edges in the product automaton.
- Relabel the falling edges of each signal by the same symbol as the rising edges of that signal. (The automaton in Fig. 3-6(f) is an incomplete dual-edge JOIN specification, resulting from relabeling after product.)
- Complete the resulting automaton to obtain a process automaton so that each action is enabled at each state.

Except for the first step, which defines the specification in terms of pulses, and the third step, which introduces additional user-defined constraints, the procedure above is automatic and can be performed by standard FIREMAPS operations. This allows users to describe pulse-mode specifications by smaller automata that ignore the interleaving of falling edges; the full transition-event specifications are then built by FIREMAPS. It is possible to add a front-end from some sort of HDL type description to facilitate the verification work, since essentially, the verification is based on state machine and netlist.

Notice that the basic idea of the method is not limited to pulse-mode specification construction, but applies in many situations where one high-level abstract event can represent several low-level signal transitions. We will introduce other such applications in our case studies.

Chapter 4. A Structured Approach for Peephole Verification

4.1. Assume-Guarantee Verification

Using hierarchical verification, a verification task $p_1 \times p_2 \sqsupseteq q$ can be split into three separate tasks that might have lower computational costs overall:

$$q_1 \times q_2 \sqsupseteq q; \quad p_1 \sqsupseteq q_1; \quad p_2 \sqsupseteq q_2. \qquad (4\text{-}1)$$

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 that the refinement between submodules and their lower level implementations, $p_1 \sqsupseteq q_1$ and $p_2 \sqsupseteq q_2$, does not always hold. p_1 and p_2 only "work correctly" in the presence of each other, thus the refinement relations $p_1 \sqsupseteq q_1$ and $p_2 \sqsupseteq q_2$ only hold under certain assumptions. In Fig. 4-1, the OR gate cannot directly refine the specification of Merge. Nevertheless, if the environment guarantees that the inputs are mutually exclusive pulses, the OR gate implements the Merge specification.

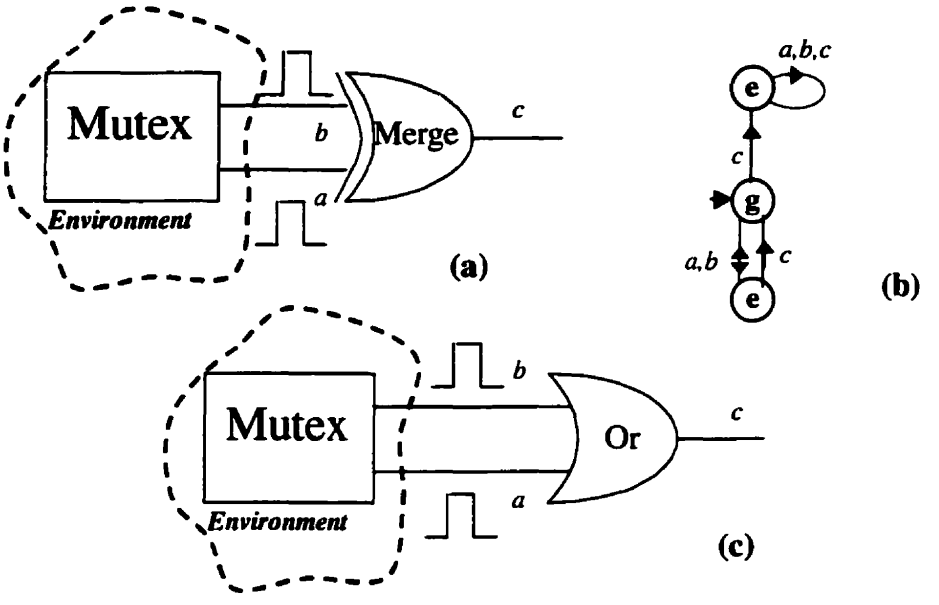


Figure 4-1: Replacement under assumptions:
(a) Merge in a certain environment; (b) Inertial XOR;;
(c) Implementation of Merge by OR under a certain environment.

Verification rules that address this difficulty are presented in [CLM89], [McM97], [HQR98], and [HQR+98]. Such verification rules are called *assume-guarantee*. In verification methods based on assume-guarantee rules, the correctness of the implementation relies on assumptions from the environment. According to assume-guarantee rules, a verification of the form $p_1 \times p_2 \sqsupseteq q$ can be decomposed as follows:

$$p_1 \times q_2 \sqsupseteq q \times q_1; \tag{4-2}$$

$$p_2 \times q_1 \sqsupseteq q \times q_2. \tag{4-3}$$

More precisely, it suffices to verify (4-2) and (4-3) for some q_1 and q_2 of a certain restricted form in order to establish $p_1 \times p_2 \sqsupseteq q$. 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 $p_1 \times p_2 \sqsupseteq q$. Usually, q_1 and q_2 are selected to have non-empty prefix-closed accessible sets to justify a structural induction argument, but other selections are also possible. In the following, we adapt the assume-guarantee rule to our applications.

4.2. The Peephole Rule

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

$$p_1 \times \ldots r_i \times \ldots \times p_n \sqsupseteq q$$

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 makes the replacement possible; and, let M be an arbitrary set of processes, call it *support model*, representing modules in the system consisting of the implementation and the environment that will guarantee the validity of the optimization assumption.

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

$$\begin{array}{lll} \textit{if} & p_1 \times \ldots p_i \times \ldots \times p_n \sqsupseteq q & \\ & \textit{and} \quad (\exists\, d \in S_E,\ M \subseteq \{p_j \mid j \neq i\} \cup \{-q\}: & \\ & \qquad r_i \times d \sqsupseteq p_i \quad \wedge \quad (\times_{m \in M} m) \sqsupseteq (\times_{m \in M} m) \times d \quad) & \\ \textit{then} & p_1 \times \ldots r_i \times \ldots \times p_n \sqsupseteq q. & \end{array}$$

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:

$r_i \times d \sqsupseteq p_i$

$\Rightarrow$ (by monotonicity w.r.t. product by $(\times_{j \neq i} p_j) \times (-q)$)

$(\times_{j \neq i} p_j) \times (-q) \times r_i \times d \sqsupseteq (\times_{j \neq i} p_j) \times (-q) \times p_i$

$\Rightarrow$ (by commutativity of product)

$$(\times_{j\neq i} p_j) \times (-q) \times r_i \times d \sqsupseteq (\times_k p_k) \times (-q) \qquad (1)$$

$(\times_{m\in M} m) \sqsupseteq (\times_{m\in M} m) \times d$ and $M \subseteq \{ p_j \mid j\neq i\} \cup \{-q\}$

$\Rightarrow$ (by montonicity of product w.r.t. components from outside M)

$(\times_{j\neq i} p_j) \times (-q) \sqsupseteq (\times_{j\neq i} p_j) \times (-q) \times d$

$\Rightarrow$ (by montonicity of product w.r.t. r_i)

$$(\times_{j\neq i} p_j) \times (-q) \times r_i \sqsupseteq (\times_{j\neq i} p_j) \times (-q) \times d \times r_i \qquad (2)$$

(1) and (2)

$\Rightarrow$ (by transitivity of refinement, since $(\times_k p_k) \sqsupseteq q \Leftrightarrow (\times_k p_k) \times (-q) \times p_i \sqsupseteq \Phi$)

$(\times_{j\neq i} p_j) \times (-q) \times r_i \sqsupseteq \Phi$

$\Leftrightarrow$

$p_1 \times \ldots r_i \times \ldots \times p_n \sqsupseteq q$

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

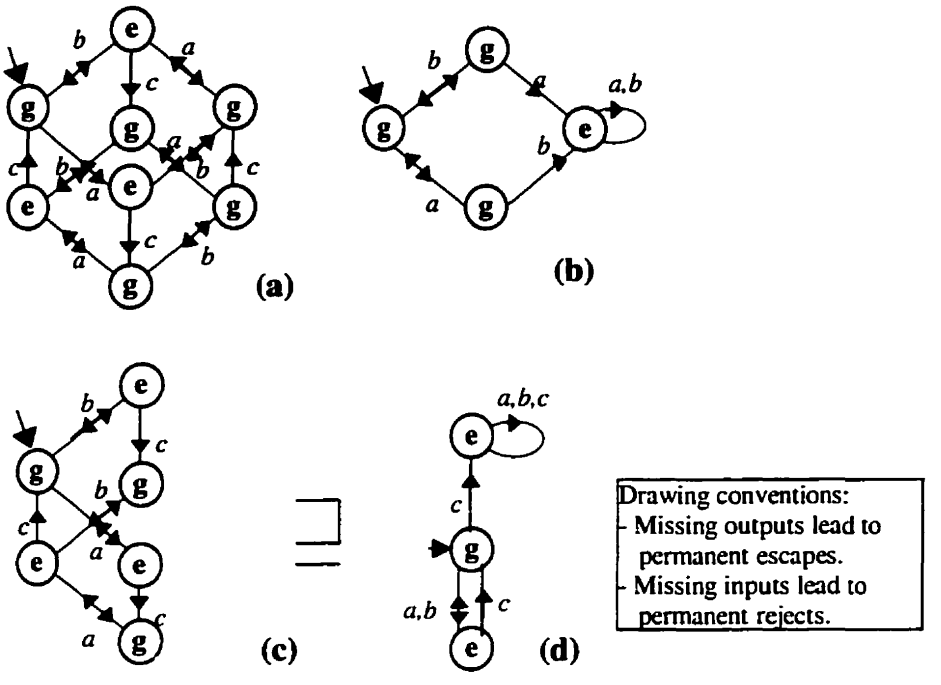


Figure 4-2: Usage of the peephole rule:
(a) Process of inertial OR gate; (b) Relative timing constraints *d*;
(c) Process of OR gate under constraints; (d) Process of merge.

Example:[1] The example in Fig. 4-1 illustrates a peephole optimization that replaces a Merge macromodule [OSC67][SS86][Su89] (essentially, an XOR gate) by an OR gate. 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.

In verification of Fig.4-1, assume refinement relation

$$p_{merge} \times \ldots p_{mutex} \times \ldots \sqsupseteq q$$

holds, in which p_{merge} represents the implementation of Merge, XOR gate, and p_{mutex} represents an existing Mutex component which physically enforces arbitration between the pulses.

[1] This example was inspired by a similar example brought to our attention by Mark Josephs.

Let $M = \{p_{mutex}\}$, and let d be the product of the relative timing constraints in Fig. 4-2 (b):

$D(a+, a-) < D(a+, b+)$;

$D(b+, b-) < D(b+, a+)$.

Because

$$p_{mutex} \sqsupseteq p_{mutex} \times d$$

and

$$p_{OR} \times d \sqsupseteq p_{merge},$$

using the peephole rule, we have:

$$p_{or} \times \ldots p_{mutex} \times \ldots \sqsupseteq q.$$

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

Theorem 1 relates to the assume-guarantee rule over a particular case. If p_1 from (4-2) is substituted by r_i from Theorem 1, q_1 from (4-2) is substituted by p_i from Theorem 1, and q_2 from (4-2) is substituted by the process $(\times_{j \neq i} p_j)$, then the hypothesis of Theorem 1 implies inequation (4-2), which is part of the hypothesis of the assume-guarantee rule.

$$r_i \times d \sqsupseteq p_i \Rightarrow r_i \times (\times_{j \neq i} p_j) \times d \sqsupseteq p_i \times (\times_{j \neq i} p_j)$$
$$r_i \times (\times_{j \neq i} p_j) \sqsupseteq p_i \times (\times_{j \neq i} p_j) = p_i \times (\times_{j \neq i} p_j) \times p_i \sqsupseteq q \times p_i$$

The remaining part of the hypothesis of the assume-guarantee rule, inequation (4-3), also follows from the hypothesis of Theorem 1 under the substitutions above, while p_2 from (4-2) can be substituted by any process p that refines $(\times_{j \neq i} p_j)$.

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 $(\times_{m \in M} m) = \Phi$. Since Φ is the most transparent process in a process space (Definition 2.13 in [Ne98]) then $d = \Phi$. In this case, repeated application of Theorem 1 matches the hierarchical verification procedure as described by inequation (4-1).

4.3. Heuristics for Finding Verification Assumptions

Not all the optimization assumptions are guaranteed by the environment of the peephole. Since Theorem 1 has no restriction 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. For example, certain delay assumptions inside a circuit can be ascertained during peephole optimization.

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. 4-2 (c), the process of "OR gate under environment assumption" is smaller than Fig. 4-2 (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 was 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.

Under certain circumstances, there are no explicit assumptions available for use in verification by our method. To reduce the verification task, we assume the relative delays of various modules are fixed and strictly ordered, and we separate the verification into several cases defined by one delay assumption each. For example, let $D1$ and $D2$ be the delays of two chains in the circuit. The verification can be separated into two cases: $D1 > D2$ and $D1 < D2$. We call this procedure *case separation*. In this example, the support model M can be part of the system that relates $D1$ and $D2$ to each other, so that the relation between them cannot alternate. (We have either $D1 > D2$ or $D1 < D2$ throughout the operation of the circuit). Formally, the optimization assumption d is the meet (alternative composition) of the chain constraint processes for $D1 > D2$ and $D1 < D2$. Since case separation can produce false passes, it should be used as a last resort. However, case separation is useful to detect flaws where other techniques fail because of computational costs.

4.4. Strategy for Assume-guarantee Verification

In this section, we give our procedure of applying assume-guarantee rules in verification. Although the following pseudo-code is developed for FIREMAPS, it should also be applicable to other tools that support refinement-based verification under relative timing constraints.

Notation:

Env_asm:	A set of timing constraints representing environment assumptions.
Des_asm:	A set of timing constraints representing design assumptions.
Ver_res:	A set of additional timing constraints detected in verifications.
p:	Implementation model.
q:	Specification model.
wit_exec:	A witness execution found in refinement checking.
wit_cc:	A relative timing constraint introduced to avoid the witness execution.

```
Ver_res=∅;
Env_asm = Initial assumptions;
Loop {
    If (p × Ver_res × Env_asm × Des_asm ⊒ q)
            Return Ver_res, Env_asm and exit;
    Else {
            wit_exec = refinement checking result;
            Case (wit_exec) {
            Can be avoided by the environment of the peephole alone:
              Create wit_cc upon wit_exec, satisfied by environment assumption;
              Env_asm = Env_asm ^ wit_cc;
            Overlaps the peephole:
              Create wit_cc upon wit_exec;
              Ver_res = Ver_res ^ wit_exec;
            }
    }
}
```

Chapter 5. Case Study: a High-Speed Arbiter

A high-speed arbiter using the asP* protocol is reported in [GO99], 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.

5.1. Verification before Optimizations

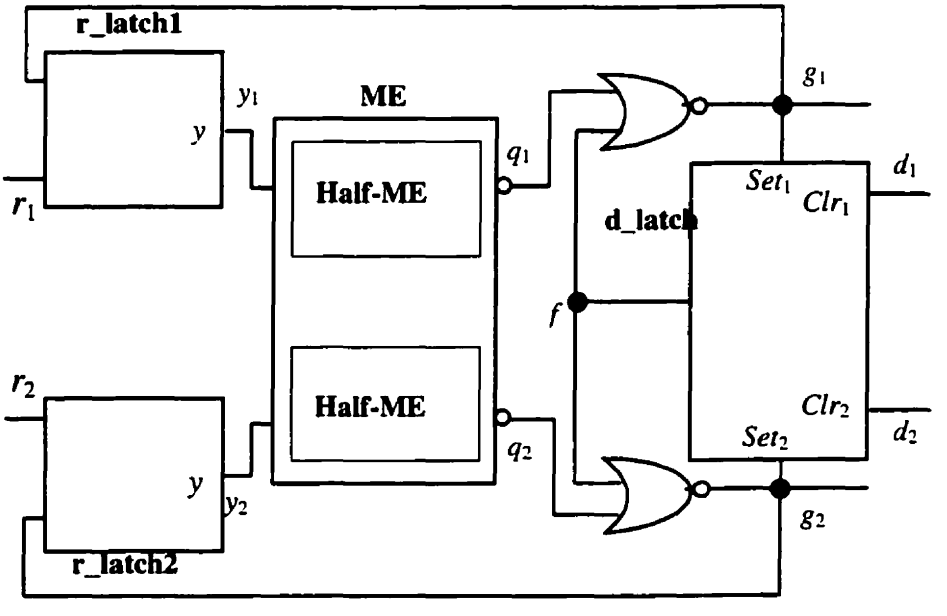


Figure 5-1: Block diagram of the asP* arbiter before optimizations.

The block diagram of the non-optimized version of the arbiter from [GO99] is shown in Fig. 5-1. 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. 5-1 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 fire the rising edge of grant1. Rising edge of g_1 is propagated to dlatch and set f high, thus reset the g_1 low; at the same time, feedback of g_1 pulse withdraws the request y_1. Pending request from channel2 gets grant from Mutex and grant g_2 to channel2 will be issued after acknowledge d_1 from channel1 is received. Notice that the grant pulse will be issued without waiting for the falling edge on the request, because the arbiter operates in single-edge mode produced only after the falling edges of the input pulses.

Notice that the events at the external interface of the arbiter are represented as pulses, but the internal signals use transition events. Moreover, certain submodules in the arbiter have different signalling conventions at their interfaces; for example, the rlatch submodule uses pulse events at the inputs and transition events at the output. For each submodule, switch level implementation is provided in [GO99]. On this flattened level, transition event is the only convention for all signals.

To prove correctness of the arbiter, we aim to establish refinement between the high-level specification and the switch-level implementation. We apply the hierarchical verification procedure described in Chapter 4, using the submodules in Fig. 5-1 as intermediate specifications. In the following, p_{Mutex}, $p_{rlatch1}$, $p_{rlatch2}$, p_{dlatch}, p_{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:

$$q_{Mutex} \times q_{rlatch1} \times q_{rlatch2} \times q_{dlatch} \times q_{nor1} \times q_{nor2} \sqsupseteq q_{arbiter}$$

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

$$p_{Mutex} \sqsupseteq q_{Mutex};\quad p_{rlatch1} \sqsupseteq q_{rlatch1};\quad p_{rlatch2} \sqsupseteq q_{rlatch2};$$

$$p_{dlatch} \sqsupseteq q_{dlatch};\quad p_{nor1} \sqsupseteq q_{nor1};\quad p_{nor2} \sqsupseteq q_{nor2}.$$

In this chapter, we call the verification on the first level high-level verification and the second level submodule verification.

5.1.1. High-level Verification

5.1.1.1. Construction of asP* Arbiter Specification

The construction of the asP* specification is shown in Fig. 5-2. The processes in Fig. 5-2 (a) specifies the behavior of the asP* arbiter by rising edges only. The processes in Fig. 5-2(b)-(g) are the pulse automata for each signal, constructed as indicated in Chapter.3. The product of all processes in Fig. 5-2 is a process describing the behavior of the asP*

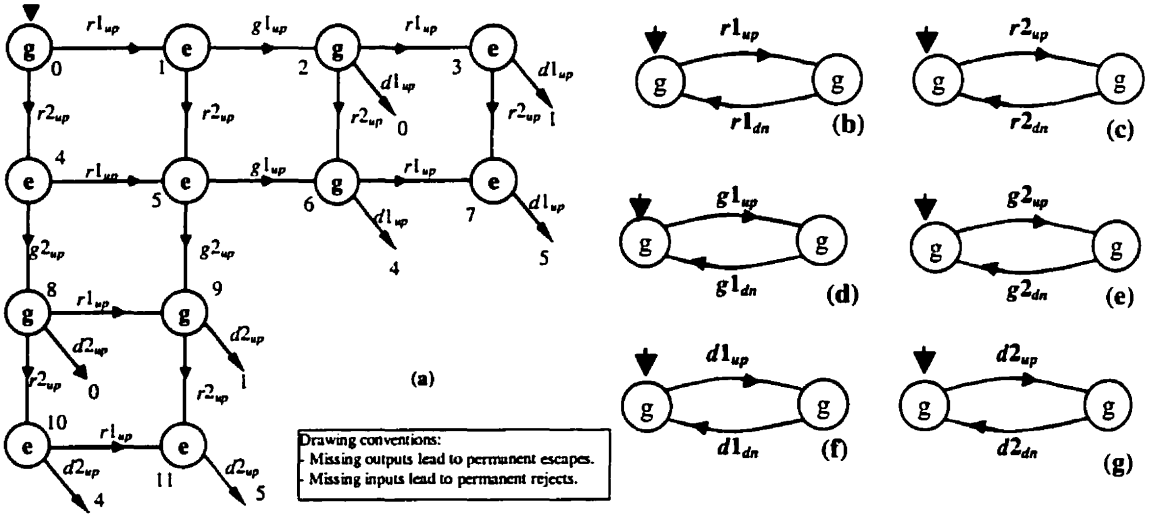


Figure 5-2: Construction of the asP* arbiter specification:
(a) Specification of asP* arbiter by rising edges only; (b)-(g) Pulse automata.

arbiter; this automaton uses transition events, with rising edges and falling edges of the same signal labeled differently. The complete transition event specification of the asP* arbiter is achieved after relabeling rising and falling edges of the same signal to the same label.

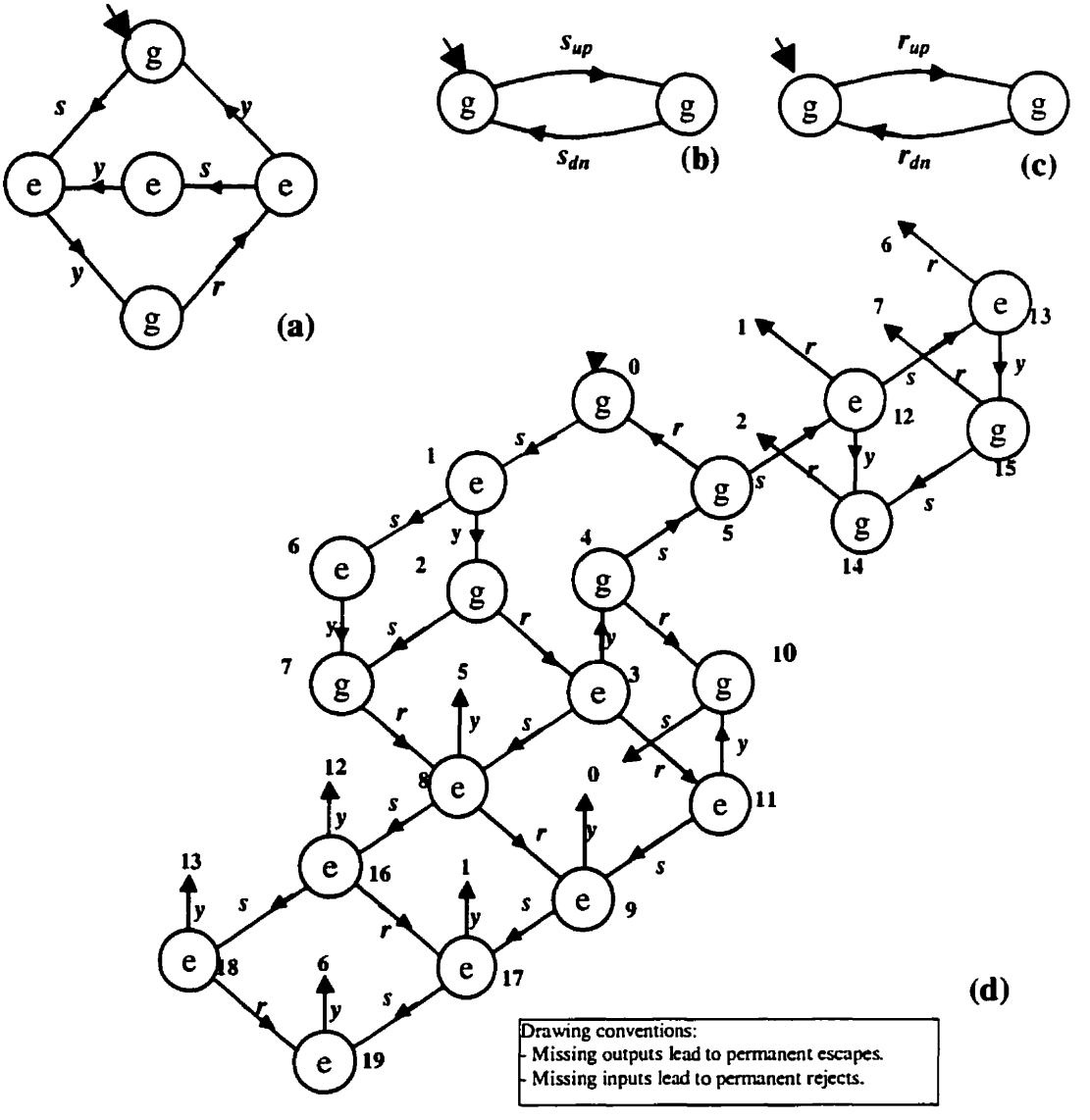


Figure 5-3: Constructing intermediate specification of rlatch:
(a) Specification of rlatch with mixed events;
(b) , (c) Pulse automata of *s* and *r*; (d) Transition-event specification of rlatch.

5.1.1.2. Implementation Description

In high-level verification, the implementation of the arbiter is described as the connection of submodules. Boolean functions are used to represent two NOR gates; 4-phase mutual exclusion model is used to describe behavior of the Mutex; and the intermediate specifications of rlatch and dlatch are constructed using the techniques introduced in Chapter 3.

Fig. 5-3 represents the construction of rlatch specification. The behavior of rlatch is defined in Fig. 5-3 (a), in which signals r and s use pulse events, and signals of y use transition events. Pulse automata are shown in Fig. 5-3 (b) and (c). Notice that only signals s and r need pulse automata. The transition event specification of rlatch is attained by, computing the product of (a), (b) and (c) together, relabeling s_{dn} and r_{dn} to s and r respectively, and filling illegal transitions of s, r and y. Fig. 5-3 (d) is the transition event specification of the rlatch without filling illegal transitions. The dlatch specification is constructed similarly.

5.1.1.3. Verification Result

A set of relative timing constraints are detected in high-level verification.

Violation 1:
A witness execution is provided by FIREMAPS as follow:

$$r_1+\ y_1+\ q_1-\ g_1+\ f+\ r_1-\ r_1+\ y_1-\ q_1+\ y_1+\ q_1-\ d_1+\ f-$$

After the rising edge of g_1 is issued, f is set to high by g_1+. At the same time, g_1+ resets the rlatch$_1$, then q_1 will be reset to high. The rising edge of q_1 or f will reset g_1 to low. If the NOR gate in the implementation is not quick enough to stabilize its output before the next request and done pulses reset q_1 and f to low, then a falling edge of g_1 is missing.

We change the NOR model to a quickened model, corresponding to the constraints below:

d_1: $D(g_i+ f+ d_i+ f-) > D(g_i+ f+ g_i-)$

d_2: $D(g_i+ q_i+ r_i+ q_i-) > D(g_i+ f+ g_i-)$ $(i = 1, 2)$

These constraints can be implemented by sizing the circuit: Make the NOR gate delay less than internal Mutex delay or the delay from when grant is issued to when done arrives.

Violation 2:

A witness execution is provided as below:

$r_2+ y_2+ q_2- g_2+ f+ g_2- d_2+ f- g_2+ f+ g_2-$

After g_2+, d_2+ resets the f to low before g_2+ sets q_2 to high. Another g pulse will be generated by f -, which is not expected by the specification.

To avoid this problem, we added several chain constraints of the following form:

d_3: $D(g_i+ f+ d_i+ f-) > D(g_i+ y_i- q_i+)$ $(i=1, 2)$

To implement these constraints, the arbiter should be sized so that the delay of rlatch plus the delay of Mutex is less than the delay of dlatch plus the delay from when grant is issued to when done signal arrives.

Violation 3:

A witness execution is provided by FIREMAPS:

$r_2+ y_2+ q_2- g_2+ f+ r_2- r_2+ y_2- y_2+ g_2-$

This execution represents a violation as follows. The g_2+ resets y_2 to low. If another request pulse r_2 comes from the same channel after g_2+, then y_2 will be set to high again. If the Mutex in implementation is not quick enough, y_2 pulse will be considered as a hazard.

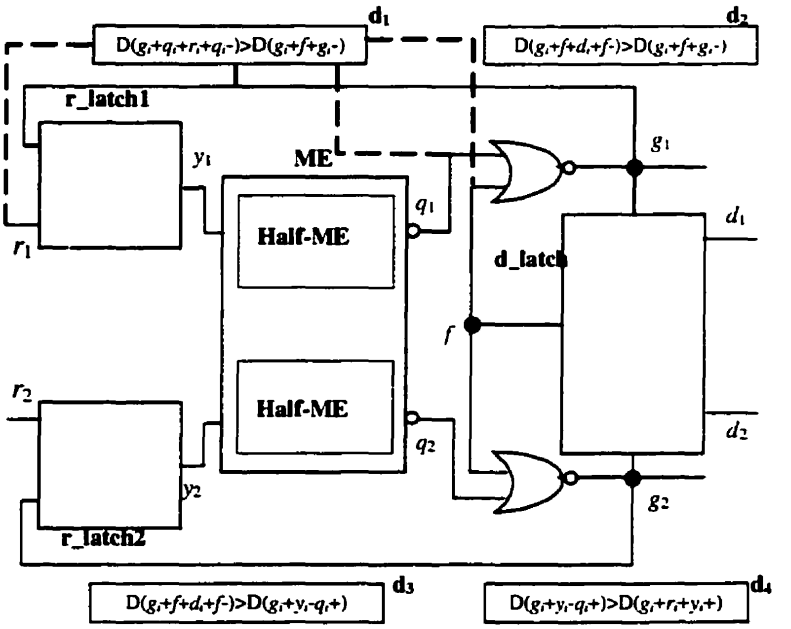


Figure 5-4: High-level verification result of the asP* arbiter.

To avoid this problem, we made additional timing assumptions by changing the Mutex from hazard intolerant model to quickened model. This change amounts to several chain constraints, of the following form:

$$d_4: \quad D\,(g_i+\, y_i-\, q_i+) > D\,(g_i+\, r_i+\, y_i+) \qquad (i=1,\, 2)$$

Notice that constraints 3 and 4 release the information that under what environment, can the arbiter work properly. To implement these constraints, the circuit should be sized so that the delay of resetting the Mutex is less than the delay from when a grant is issued to when the next request arrives from the same channel.

Fig. 5-4 shows the high-level verification result of the asP* arbiter:

$$q_{Mutex} \times q_{rlatch1} \times q_{rlatch2} \times q_{dlatch} \times q_{nor1} \times q_{nor2} \times d_h \sqsupseteq q_{arbiter}$$

$$d_h = d_1 \times d_2 \times d_3 \times d_4$$

In Fig. 5-4, boxes $d_1 - d_4$ represent 4 relative timing constraints. These constraints are modeled by processes. Product operation connects delay constraints to other submodules.

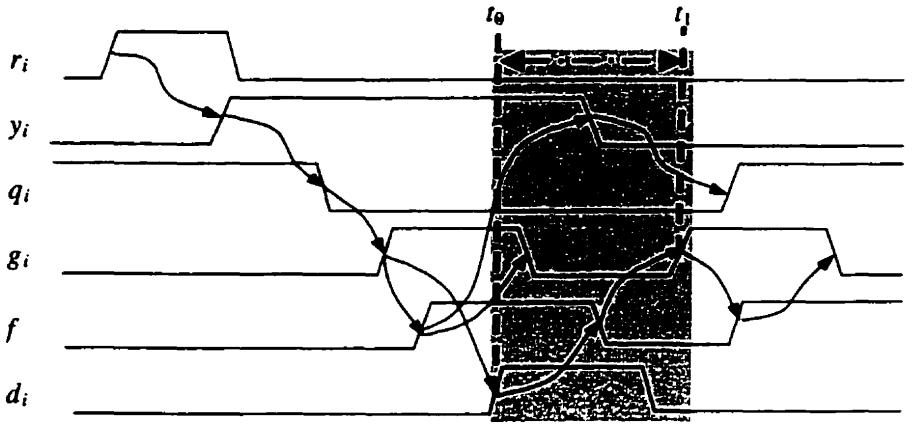


Figure 5-5: Witness execution analysis.

For example, dash lines in Fig. 5-4 connect d_1 to the corresponding signals in the circuit. For simplicity, other connections between relative timing constraints and circuit are omitted in Fig. 5-5.

Given a witness execution, we are interested to determine the corresponding circuit flaw and a delay constraint to avoid it. Such witness execution analysis is based on the waveform-like trace provided by FIREMAPS. Fig. 5-5 represents the waveform of Violation 2. The shade in Fig. 5-5 highlights the occurrence of the violation. Event d_i+ at t_0 triggers the violation; at t_1 the violation is observed. Waveform before t_0 can be considered as "path sensitization" for the witness execution. Sometimes, the path sensitization contains a large number of events, which are not very relevant for the actual violation. In the rest of this thesis, we omit representing the path sensitization portion of a witness execution, to focus on the actual violations.

Delay constrains are drawn from the witness execution analysis. The event that caused the witness execution to exit the set of legal behaviors is a good candidate for the end of the short chain or the long chain in a chain constraint: relative timing should ensure that event occurs earlier or later, when it becomes safe. Presently, we still manually generate the relative timing constraints, because the relative timing constraints are often ad hoc to the design. Some heuristics for finding such constraints are indicated in [NP98].

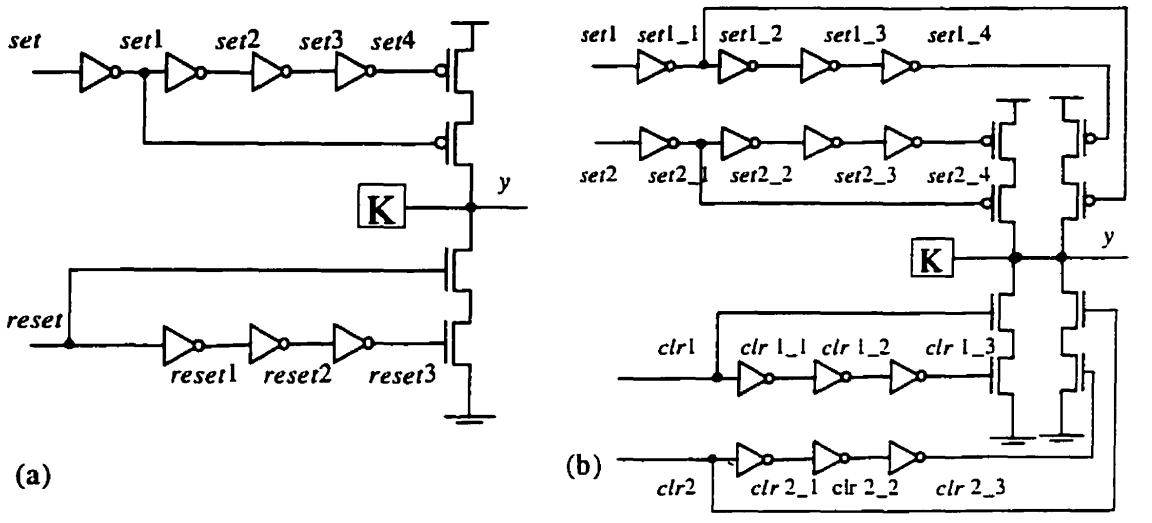


Figure 5-6: Implementation of rlatch and dlatch:
(a) rlatch implementation; (b) dlatch implementation.

5.1.2. Submodule Verification

Submodule verification consists of refinement checks between switch level implementation of submodules and their intermediate specifications. In this section, we use rlatch as the example of refinement checking on this level.

5.1.2.1. rlatch implementation

The implementation of the rlatch from [GO99] is shown in the Fig. 5-6 (a). When a rising edge of signal *set* comes, the inverter chain of set (contains 4 inverters) generates a three-inverter delay pulse upon the PMOS network, thus set output *y* high. The same mechanism is used for reset. The box labeled *K* is a "keeper" circuit consists 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.

Fig. 5-6 (b) shows the implementation of dlatch. Essentially, it is the composition of two rlatches that share the "keeper".

5.1.2.2. Switch-level Verification

We follow the approach to switch level verification reported in [Ne98], in which MOS transistor networks are decomposed into *channel-connected subnetworks* [Br87], which are defined by the absence of gate-drain and gate-source connections.

Normally each channel-connected subnetwork contains an N-transistor network and a P-transistor network, called the *pull-down* and the *pull-up* networks. Boolean variables are used to indicate the presence of conducting paths. A *NODE model* [Ne98] is used to connect the pull-down model and pull-up model together.

Figures in Fig. 5-7 show NODE symbol and models introduced in [NP98]. Fig. 5-7 (a) is the symbol of the node. Signal *up* and *dn* represent the connection to the source and ground respectively, with "1" standing for connected situation. Fig. 5-7 (b) describes the behavior of a NODE model. For example, from initial state, the transition *up* pulls the output *y* of the node to high. This model also implies that *up* and *dn* should not hold in the same value. When both *up* and *dn* are high, the MOS network is in short status, and when both *up* and *dn* are low, the MOS network is in float status. NODE model can be changed to represent the different behavior of the node under different assumptions. For example,

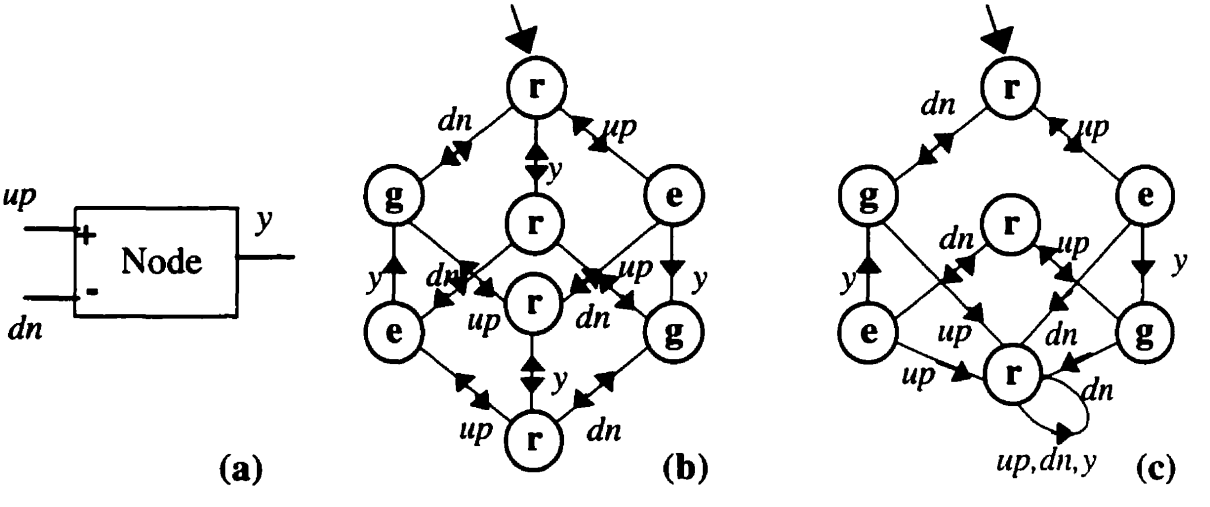


Figure 5-7: Node models :

(a) Symbol of Node; (b) Common node model; (c) Node with transient collisions illegal.

an alternative NODE model would be to make transient collision illegal by taking all executions that pass through a collision state to be rejects, as in Fig. 5-7 (c).

We modified the model of NODE by adding Keeper. As it is shown in Fig. 5-8 (a), output *y* will keep the previous value and will not dangle when none of pull-up and pull-dn network is conducted. (In Fig, 5-8 (b), by modifying reject states, in which *up* and *dn* are both low, to goal states and removing edges between two states.) The representative of pull-up and pull-down networks in the rlatch is shown in Fig. 5-8 (c), represented by Boolean functions:

$$up = \neg set1 \wedge \neg set4$$

$$dn = reset \wedge reset3$$

The verification of rlatch is to check the refinement relation as follow:

$$p_{rlatch} = p_{set\_inv\_chain} \times p_{reset\_inv\_chain} \times p_{up} \times p_{dn} \times p_{Knode} \sqsupseteq q_{rlatch}$$

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_{up} and p_{dn} are pull up and pull down functions of the node; p_{Knode} is the model of the node with keeper represented in Fig. 5-8 (b), and

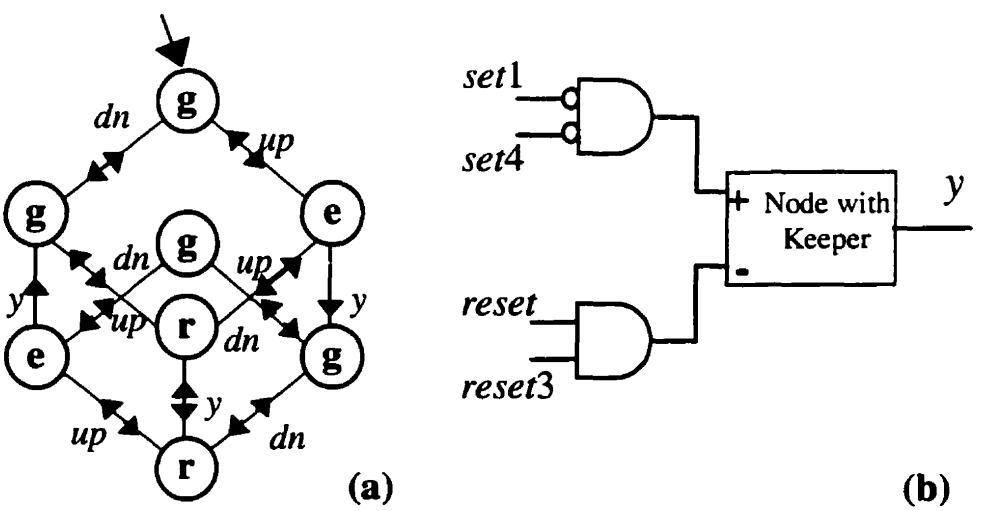


Figure 5-8: Model of node with Keeper.
(a) Node with Keeper; (b) MOS network models of rlatch.

specification q_{rlatch} is constructed in Section 5.1.1.2.

Some assumptions can be drawn from the environment of the rlatch in asP* arbiter system. Interface signal *set*, *reset* and *y* of the rlatch are under constraints. For instance, in the channel$_1$ of asP* arbiter, grant signal g_1 is used as reset signal of rlatch$_1$. Notice that for rlatch$_1$, 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. (This assumption is not the only possible such assumption.) Represented by the relative timing constraint, this environment assumption is shown as follow:

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

Correspondingly, the verification of the rlatch becomes:

$$d_{env} \times p_{set\_inv\_chain} \times p_{reset\_inv\_chain} \times p_{up} \times p_{dn} \times p_{Knode} \sqsupseteq q_{rlatch}$$

The verification result shows that the refinement relationship holds when the input pulse width is enough to be caught by the inverter chain. Since this can be easily satisfied in design, rlatch implementation in Fig. 5-8 (a) refines rlatch specification when it is used in asP* arbiter.

The same as rlatch, Mutex, dlatch and NOR gates refine their intermediate specification when they are used in asP* arbiter.

5.2. Verification of the Peephole-Optimized of asP* Arbiter

To achieve higher performance, the designers optimized the asP* arbiter of Fig. 5-1. The optimization weakens the modularity of asP* arbiter, thus increases difficulty of verification by generating more complex submodule interfaces. In this section, we introduce our experiment of applying peephole rules in peephole verification.

5.2.1. Peephole Optimizations of the asP* Arbiter

Fig. 5-9 (a) is the block diagram of asP* arbiter after optimization in [GO99]. Notice the change of submodules at their interfaces. (For example, the new interface of Mutex has 12 signals.) In Fig. 5-9 (a), signal r_2 is connected to the internal node set_2 of rlatch; signal g_b is connected to the internal node set_1 of dlatch.

Fig. 5-9 (b) shows the peephole optimizations over the half Mutex. Boxes in dark indicate modifications during the optimizations. Signal r_2 and g_b from the internal nodes of rlatch

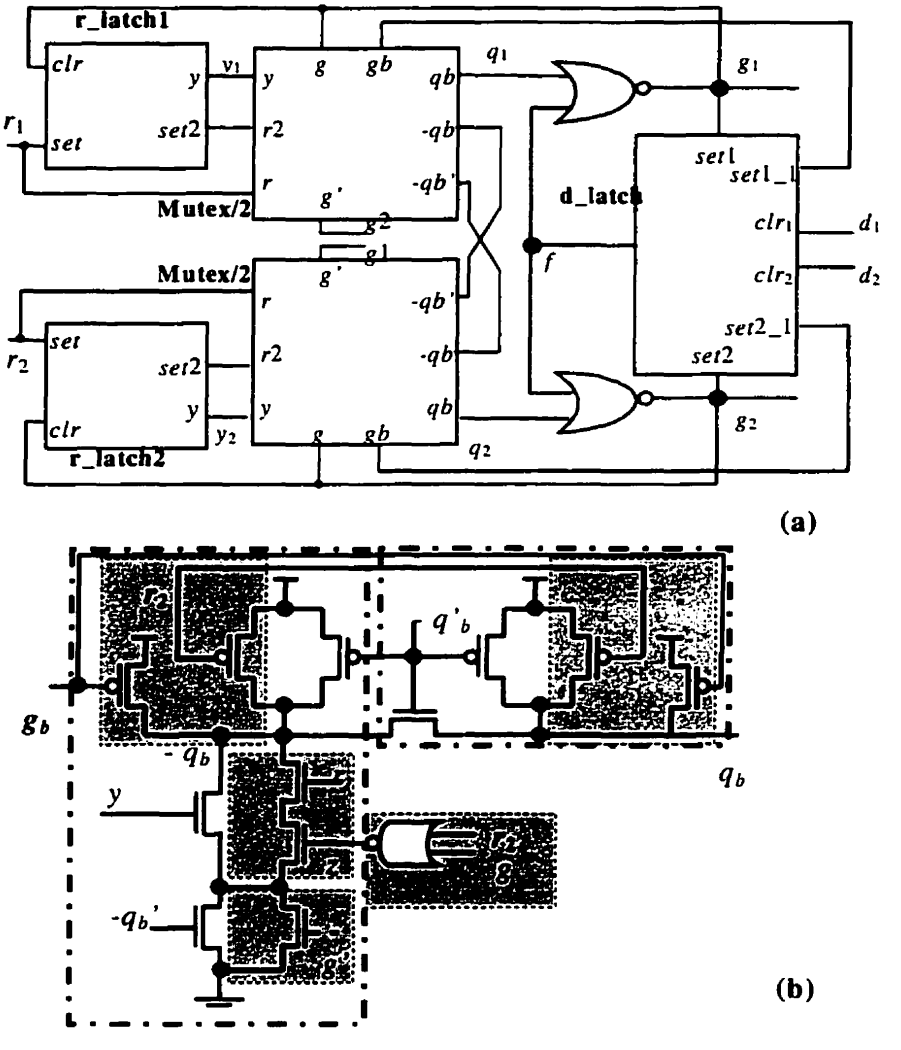


Figure 5-9: Peephole Optimization of asP* arbiter:
(a) asP* arbiter block diagram after optimization; (b) Half-Mutex after optimization.

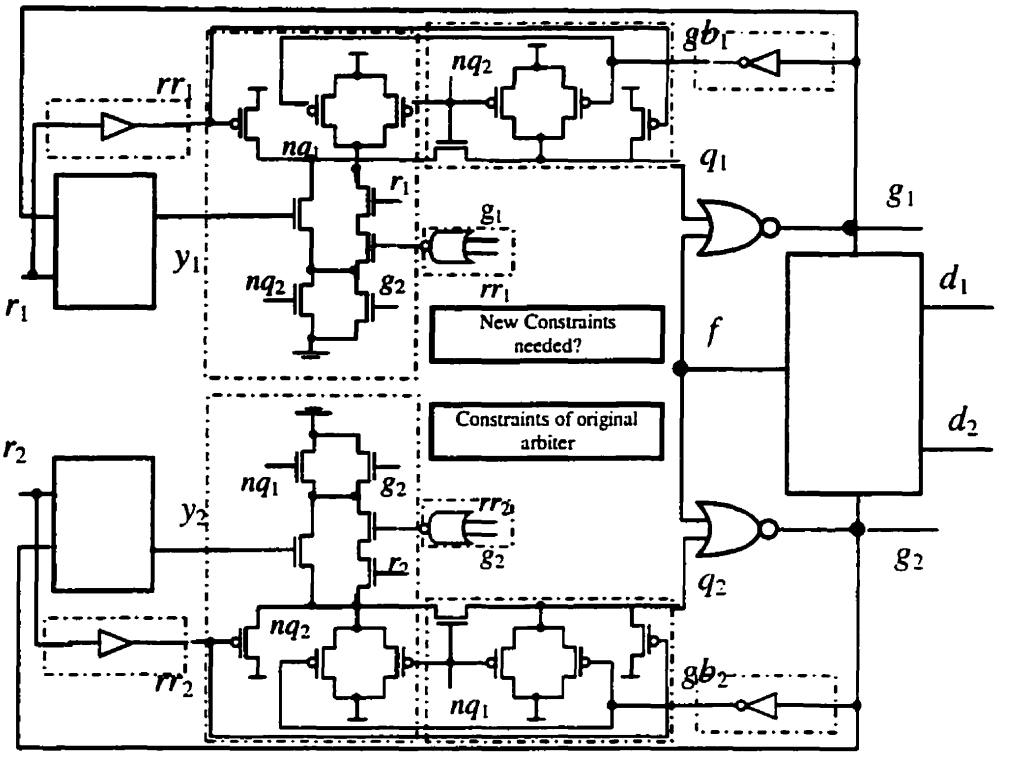


Figure 5-10: asP* arbiter after optimization.

and dlatch are highlighted. For the detail and motivation of these optimizations, we refer the reader to [GO99].

Notice that only Mutex's low-level structure is changed by optimization. For the other submodules, optimization only changes the topology between them.

To verify the arbiter after peephole optimization, we model the optimized arbiter, shown in Fig. 5-10. Model rr1, rr2, gb1 and gb2 are "abstracted" from rlatch1, rlatch2 and dlatch. Relations between models hold as follow:

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

In this way, we isolate unchanged modules (white blocks) in Fig. 5-10 from the models changed by optimization (dark blocks). Verification task after peephole optimization verification is described as follow:

Reusing the result from Section 5.1 we want to check refinement relation:

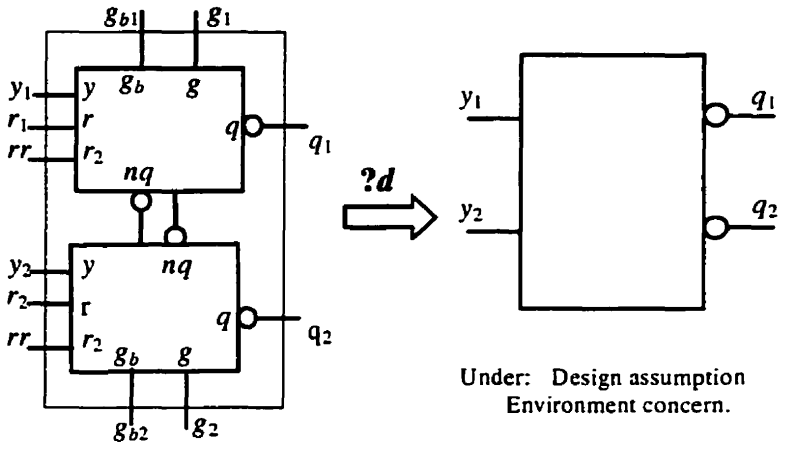


Figure 5-11: Reducing verification complexity by using the peephole rule.

$$p_{gb1} \times p_{gb2} \times p_{rr1} \times p_{rr2} \times p_{Mutex'} \times q_{rlatch1} \times q_{rlatch2} \times q_{dlatch} \times q_{nor1} \times q_{nor2} \sqsupseteq q_{arbiter}$$

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

5.2.2. Verification of Peephole Optimizations

Applying peephole rule in verification, the verification of arbiter can be reduced to verifying the relation

$$p_{Mutex'} \times d \sqsupseteq q_{Mutex} \qquad ? d$$

as it is shown in Fig. 5-11. If constraint d does not impose extra confinement to the system, the replacement of Mutex is successful. Otherwise extra constraints in d should be examined.

There are three optimizations addressed in [GO99]. The first optimization over the half-Mutex is shown in Fig. 5-12. Changes involved in optimization1 are highlighted.

To apply the peephole rule, the preparation work consists of three steps:

First, a model of optimization assumption is constructed. In Fig. 5-12 the highlighted part indicates the optimization assumption of optimization1, 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." [GO99]. Relative timing constraint

$$d_1: \quad D(r_i+, nq_i-) > D(r_i+, y_i+) \qquad (i = 1, 2)$$

in Fig. 5-12 comes from this statement.

Second, the refinement

$$p_{gb1} \times p_{gb2} \times p_{rr1} \times p_{rr2} \times q_{Mutex} \times q_{rlatch1} \times q_{rlatch2} \times q_{dlatch} \times q_{nor1} \times q_{nor2} \times d_1 \sqsupseteq q_{arbiter}$$

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

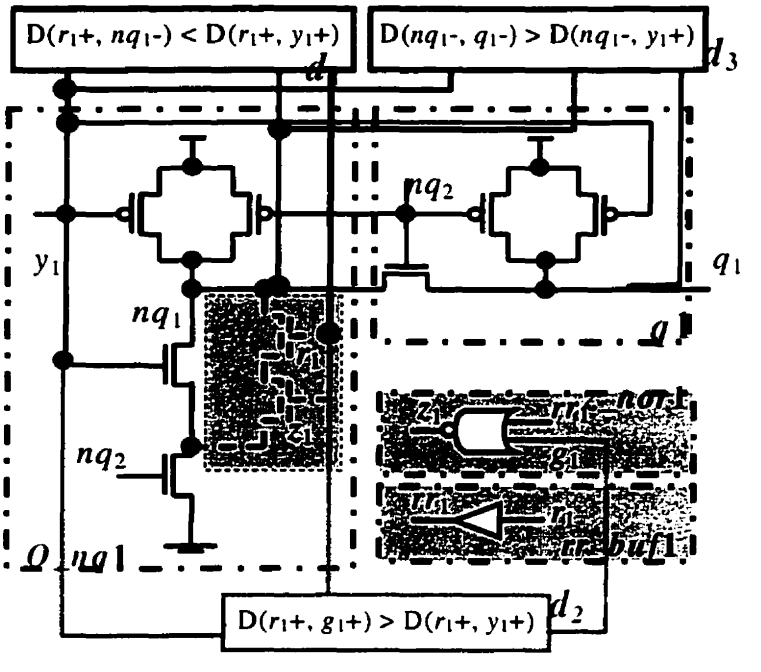


Figure 5-12: Verification and result of optimization1 (half).

peephole verification, and optimization assumption $d = d_1 \times p_{rr1} \times p_{rr2}$.

The verification procedure follows the strategy in Chapter.4. Two constraints are introduced in the procedure. One constraint:

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

can be supported by component model $p_{rlatch1}$ and $p_{rlatch2}$. The result of peephole verification is:

$$M = \{ p_{gb1}, p_{gb2}, p_{rr1}, p_{rr2}, d_1. q_{rlatch1}, q_{rlatch2}); d = d_1 \times d_{rr1} \times d_{rr2} \times d_2$$

$$p_{Mutex'} \times d \times d_3 \sqsupseteq q_{Mutex}$$

In which d_3 is an extra delay constraint:

d_3: $D(nq_i-, q_i-) > D(nq_i-, y_i-)$ $(i = 1, 2)$

needed for the holding of the refinement.

The result implies that: if the "short path" of r_1 does not change the order of y_1+ and q_1, after optimization1, arbiter implementation still refines the specification.

Optimization2 and optimization3 are verified in the same way and the refinement only holds under certain extra constraints.

To examine the validity of the extra 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. 5-10. The verification terminated and the result reported that no extra relative timing constraints are needed for arbiter optimization:

$$M = \{d_{des1}, d_{des2}\}; d = d_{des1} \times d_{des2}$$

$$p_{Mutex'} \times q_{rlatch1} \times q_{rlatch2} \times q_{dlatch} \times q_{nor1} \times q_{nor2} \times d_h \times d \sqsupseteq q_{arbiter}$$

d_{des1}: $D(r_i+, nq_i-) < D(r_i+, y_i+)$ from optimization1

d_{des2}: $D(g_i+, g_{bi}-) > D(g_i+, y_i+)$ from optimization2

$(i = 1, 2)$

So the optimized arbiter in Fig. 5-9 (a) can replace the arbiter in Fig. 5-1.

Chapter 6. Case Study: Communication Refinement

Communication refinement is an important technique for reducing the design effort for system on chip architectures [RSV97]. The case study of this chapter is to verify one step of communication refinement using modules that interface locally clocked domains to a global (across-chip) handshake environment. The modules were proposed by [MVK+99, MVF00], following the pausible clock idea of [YD96].

Just like in the pulse-mode specifications, we start with a simpler specification for data transfer that uses high-level events, then we construct a full-detail specification by product with constraint processes. A GALS wrapper implementation, as a particular implementation of data communication, is regarded as a special kind of peephole optimization, applying the peephole rule.

6.1. Data Transfer Specification

6.1.1. Communication Features.

Essentially, communication among blocks in a system 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 example we use in this chapter 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.

Fig. 6-1 represents the data transfer implementation. A data channel should be implemented to connect sender and receiver, so that data sent by the sending side (*Data*1) can propagate to the receiving side (*Data*2); control signals *data_snd* and *data_rev* synchronize the behavior of sender, receiver and data channel. The main difference between synchronous and asynchronous data transfer is how control signals synchronize the sender, receiver and data channel in communication.

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

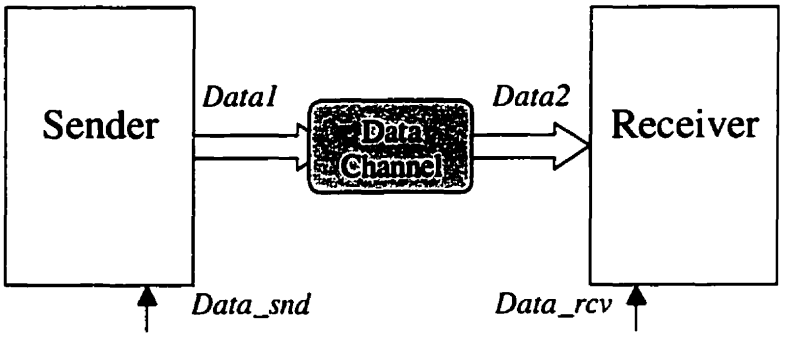


Figure 6-1: General data communication diagram.

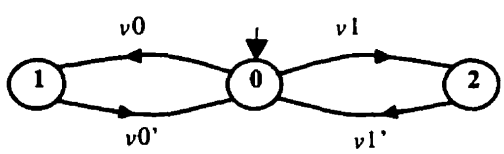
- Data integrity: received data preserves its original sending value.
- Stream integrity: no data items are lost or duplicated during data transmission, and the order of data items is preserved through the transfer.

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 on the sending side, there is a data validity event at the input port of the data channel.
- After data is properly sampled by 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.

For the asynchronous wrapper, the high-level data-validity specification is the process shown in Fig. 6-2. The "0" data validity event *v*0 is propagated as *v*0' to the end of data channel. Same applies for the "1" data validity events. Notice that we use only one-bounded models for the data propagation. An N-bounded model is easy to achieve following the example of constructing a two-bounded buffer, which is introduced in



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.

Figure 6-2: Propagation of validity events.

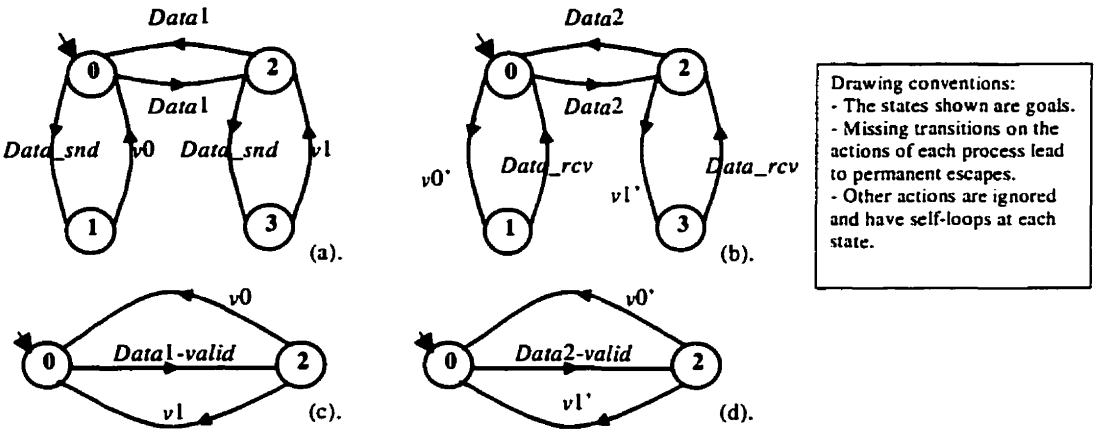


Figure 6-3: Examples of Fusion Processes:
(a). Fusion of *Data*1; (b) Fusion of *Data*2;
(c) Invariant fusion of *Data*1; (d) Invariant fusion of *Data*2.

Chapter 2.

6.1.2. Fusion Processes

In order to transform the high-level specification of Fig. 6-2 into a full-blown transition-event specification, 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. 6-3 (a) illustrates a fusion process where the data transition event *Data*1 is fused with its validity events *v*0 and *v*1 by control event *Data_snd*, which is the active edge of a local clock. Note that signal *Data*1 can toggle arbitrary in a clock cycle, while the validity events are related to the logical level that signal *Data*1 has right before active-edge of *Data_snd*: If *Data_snd* comes when *Data*1 is low, validity event *v0* will be issued and there will be no another validity event until the next active edge of *Data_snd*, though *Data*1 might keep changing between two active clock events. Same as Fig. 6-3 (a), Fig. 6-3(b) fuses the transition events of *Data*2, with the active edges of control signal event *Data_rcv* and with corresponding validity events. Fig. 6-3 (c) and (d) are fusion processes to glue the validity events with a higher-

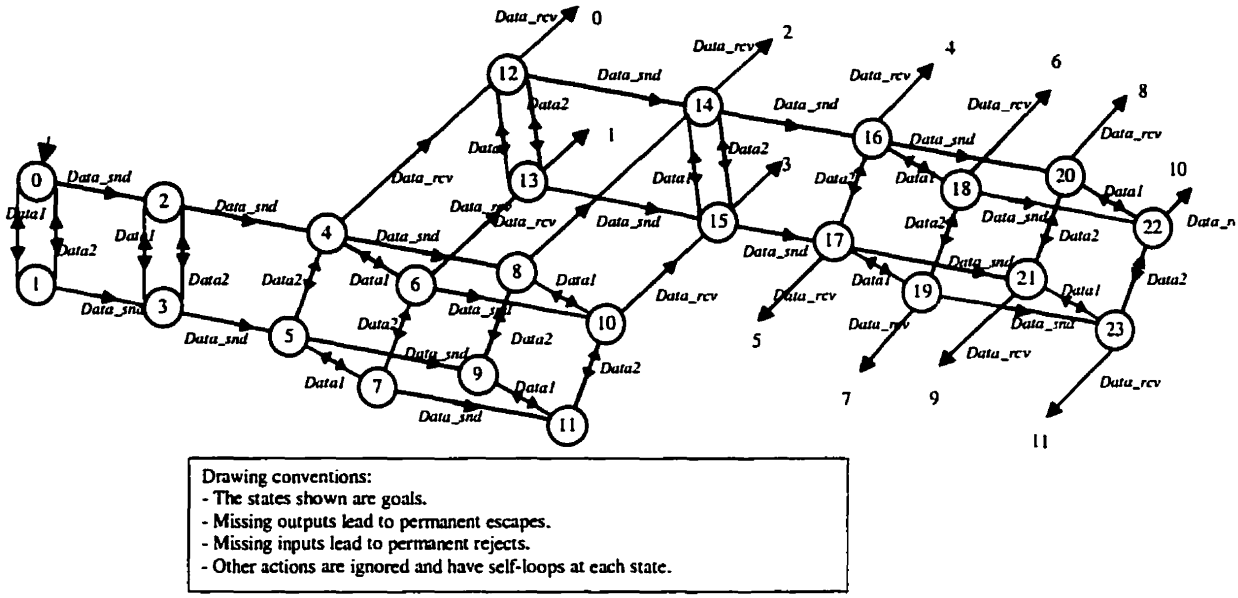


Figure 6-4: Specification constructed from fusions.

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. 6-4 a state diagram for the process obtained by applying fusion process to "glue" both *data*1 and *data*2. To obtain the full-blown transition-event specification, we first compute the product of the high-level specification in Fig. 6-2 with the fusion events in Fig. 6-3 (a) and (b), then by hiding the high-level data-validity events, we get the complete specification of data transfer, but denoted by abstract events. After applying the technique introduced in Chapter 3, we get the transition event specification of the data transfer, which can be directly used in verification.

6.2. GALS Wrapper Verification

6.2.1. GALS Wrapper

Globally-Asynchronous Locally-Synchronous architectures (GALS) 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, [BC96], [MVF00] and [MVK+99] 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 [MVF00].

The asynchronous wrapper circuits proposed in [BC96] and [MVK+99] attempted to realize failure-free communication in presence of metastability by performing arbitration between local clocks and handshaking control signals. Fig. 6-5 shows one configuration from [MVK+99]. We refer readers to [MVK+99] and [MVF00] for the detail of wrapper implementation. The gray box in center contains the implementation of the GALS wrapper. Compare with Fig. 6-1, signals *Data*1, *Data*2, *lclk*1 and *Ts* at the interface (the boundary of gray box in Fig. 6-5) of GALS wrapper correspond to signal *Data*1, *Data*2,

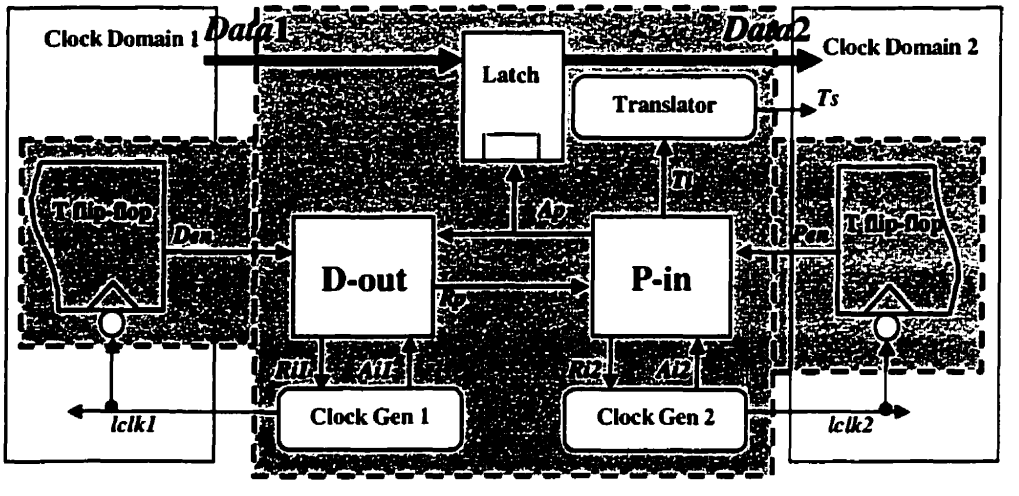


Figure 6-5: Data channel between two independently clocked domains.

Data_snd and *Data_rcv*. Signal *lclk2* is an "extra" signal. Two gray boxes next to GALS wrapper are used to construct part of support model in verification. Support model *M* also contains submodule *Translator* inside the wrapper, in which relation between signals *Ts*, *Ti* and *lclk2* is specified. Therefore, constraint *d* can be abstracted from support model *M*. The high-level verification is performed by selecting wrapper as the peephole in the communication system.

6.2.2. High Level Verification Result

Here we verify a step of communication refinement by checking whether our data-transfer specification in Fig. 6-4 is satisfied by the channel configuration in [MVK99]. We refer reader to [KNY01] for intermediate specifications of submodules. 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 absence of deadlock and unfairness conditions 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. Only relative timing constraints are represented as chain constraints, (Signal labels refer to Fig. 6-5.) because the complexity of the witness executions.

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 [9], there is no mention 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 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 up a delay time boundary for 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 will 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 close enough to *lclk2+*, then, *Ri2+*, which was supposed to be triggered by both *Rp+* and a *Pen* event (*Pen+/-*) can not be 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*'s state will be reset. Thus, *Ti* event would be canceled, before the *Ts* is set to high. While the acknowledge *Ai*2 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 *Ts*+ event. If clock2 pause request (*Ri*2) can not withdraw before *Pen*- comes, *Ti* will be reset to low again and still might affects event state of *Ts*. We add the above constraint, which implies that delay from *lclk*2- to *Pen*-, is longer than the delay from *Ai*2+ to *Ri*2-, so that no data will be missed during transfer.

Result of GALS wrapper verification is represented as follow:

$$d_{ver\_res} \times q_{C\text{-}gen1} \times q_{D\text{-}out} \times q_{P\text{-}in} \times q_{C\text{-}gen2} \times d \sqsupseteq q_{datar\_com}$$

in which d_{ver_res} is the product of all relative timing constraints detected in verification.

Chapter 7. Concluding Remarks

We introduce two techniques to solve difficulties encountered in formal verification: specification construction and state explosion. The specifications construction technique bridges the specifications on different levels: high-level analysis and low-level verification. We provide a structured verification rule, called peephole rule, for reducing verification complexity in peephole optimizations. Just like hierarchical verification and assume-guarantee rules, the peephole rule alleviates state explosion by splitting an overall proof obligation into several smaller verification tasks.

We demonstrate our techniques by applying them to different circuits from [GO99] and [MVK+99]. The circuits in [GO99] were previously thought to be hard to verify case studies, which are thought hard to verify; in fact, [GO99] mentions that verifications using Mocha [AHM+98] were not successful to the date of publication.

We have established refinement between the switch-level implementation of the asP* arbiter and a specification derived automatically from high-level handshake events. We found the circuit to be generally in agreement to its specification, although the delay constraints were only partly documented in [GO99].

For the GALS wrapper in [MVK+99] and [MVF00], we have found several race conditions that could not be justified by the explanations of the delay constraints given in their original papers.

References

[AG92] V. Akella and G. Gopalakrishnan, "SHILPA: A high-level synthesis system for self-timed circuits," in *Int. Conf. Computer-Aided Design, ICCAD'92*, Nov. 1992, pp. 587–591.

[AHM+98] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran, "MOCHA: modularity in model checking," *Computer-Aided Verification (CAV 98)*, pp. 521-525, 1998.

[BC96] D. Bormann, P. Cheung, "Asynchronous Wrapper for Heterogeneous Systems." *Proceeding of International Conference on Computer Design (ICCD)*, pp. 1996.

[BCM92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. "Symbolic model checking: 10^{20} states and beyond." *Information and Computation*, 98(2), June 1992.

[Be93] K. van Berkel. "Handshake Circuits: an Asynchronous Architecture for VLSI Programming." In *volume 5 of International Series on Parallel Computation.* Cambridge University Press, 1993.

[Br87] R.E. Bryant. "Boolean Analysis of MOS Circuits." *IEEE Transactions on Computer-Aided Design*, 4: pp. 634--649, July 1987.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 40-45, 1990.

[BS89] E. Brunvand and R. F. Sproull, "Translating concurrent programs into delay-insensitive circuits," in *Int. Conf. Comput. Design*, Nov. 1989, pp. 262–265.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. "Model checking and abstraction." *Proceedings of the Symposium on Principles of Programming Languages*, pp. 343-354, 1992.

[CES86] E.M. Clarke, E. A. Emerson, and A.P. Sistla. "Automatic verification of finite state concurrent systems using temporal logic specifications." *ACM transactions on Programming Languages and Systems*, 8(2) pp:244-263 (April 1986)

[CLM89] E. M. Clarke, D. E. Long, K. L. McMillan. "Compositional Model Checking." *Proceedings of Fourth Annual Symposium on Logic in Computer Science.(LICS '89)*, pp. 353-362, 1989.

[Di89] D. L. Dill. *Trace teory for Automatic Hierarchical Verification of Speed-Independent Circuits*. An ACM Distinguished Dissertation. MIT press, 1989.

[Go99] G. Gopalakrishnan. "Peephole Optimization of Asynchronous Macromodule networks." *IEEE transitions on very large scale integration (VLSI) system, vol. 7*, NO. 1, March 1999.

[GO99] M.R. Greenstreet, T. Ono-Tesfaye. "A fast, asP* RGD arbiter." *Proceedings of the Fifth International Symposium on Advanced Research on Asynchronous Circuits and Systems*, pp. 173-85, 1999.

[Ha95] S. Hauck. "Asynchronous design methodologies: an overview." *Proceedings of the IEEE*, Vol. 83, pp: 69-93, 1995.

[HBP+93] J. Haans, K. van Berkel, A. Peeters, and F. Schalij. "Asynchronous multipliers as combinational handshake circuits." *Proceedings of IFIP Working Conf. Asynchronous Design Methods*, Manchester, U.K., Mar. 31–Apr. 2, 1993.

[Ho85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HQR98] T. Henzinger, S. Qadeer, S. Rajamani, "You assume, We Guarantee: Methodology and Case Studies." *Proceeding of the International Conference on Computer-aided Verification (CAV)*, pp. 440-451, 1998.

[HQR+98] T. Henzinger, S. Qadeer, S. Rajamani, S. Tasiran, "An assume-Guarantee Rule For Checking Simulation." *Proceeding of the second International Conference on Formal Methods in Computer-aided Design (FMCAD)*, pp. 421-432, 1998.

[KCK+99] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, A. Yakolev. "Automatic Synthesis and Optimization of Partially Specified Asynchronous Systems." *Proceedings of the International Conference on Design Automation Conference (DAC99)*, pp.110 –115, 1999.

[KN01a] X. Kong, R. Negulescu. "Formal Verification of Pulse-Mode Asynchronous Circuits". *Proceedings of the International Conference on Asia South Pacific Design Automation Conference.* (ASP-DAC 2001) pp. 347-352. 2001.

[KN01b] X. Kong, R. Negulescu. "Formal Verification of Peephole Optimization". *Proceedings of 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE01)*, pp 219-234.

[KNY01] X. Kong, R. Negulescu, Larry Ying. "Refinement-based Formal Verification of Asynchronous Wrappers for Independently Clocked Domains in Systems on Chip". *The 11th Advanced Research Working Conference on Correct*

Hardware Design and Verification Methods (CHARME 2001). (Paper to be appear)

[LGS+95] 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.

[LL90] L. Lamport and N. Lynch. "Distributed computing: Models and methods." In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B, Formal Methods and Semantics*, pages 1159-1196. The MIT Press-Elsevier, 1990.

[Mc93] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[McM97] K.L. McMillan. "A compositional rule for hardware design refinement." *Proceedings of Computer-Aided Verification(CAV97)*, Lecture Notes in Computer Science 1254, pages 24-35. Springer-Verlag, 1997.

[MJCL97] C. E. Molnar, I. W. Jones, B. Coates, and J. Lexau. "A FIFO ring oscillator performance experiment." *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1997.

[MVF00] J. Muttersbach, T. Villiger, W. Fichtner. "Practical Design of Globally-Asynchronous Locally-Synchronous Systems." *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems,* 2000.

[MVK+99] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, W. Fichtner, "Globally-Asynchronous Locally-Synchronous Architectures to Simplify the Design of On-Chip Systems." *Proceedings of ASIC/SOC Conference*, pp. 317 -321, 1999.

[Ne98] R. Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits*. PhD thesis, University of Waterloo, 1998.

[Ne00] R. Negulescu. "Process spaces." *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*, pp. 196-210, 2000.

[NP98] R. Negulescu, A. Peeters. "Verification of speed-dependences in single-rail handshake circuits." *Proceeding of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp.159-170, 1998.

[OSC67] S. M. Ornstein, M. J. Stucki, and W. A. Clark. "A functional description of macromodules." In *Spring Joint Computer Conf.*, AFIPS, 1967.

[PCKP00] M. A. Peña, J. Cortadella, A. Kondratyev, and E. Pastor. "Formal verification of safety properties in timed circuits." *Proceedings of. International Symposium on Advanced Research in Asynchronous Circuits and Systems ASYNC'2000*, pp 2-11, 2000.

[Pe96] Ad M. G. Peeters. *Single-rail handshake circuits*. Ph. D. thesis, Eindhoven University of Technology, June 1996.

[PU98] L.A. Plana, S. H. Unger, "Pulse-mode macromodular systems," *Proceedings of Computer Design: VLSI in Computers and Processors ICCD '98*, pp. 348 –353, 1998

[Ra96] Jan M. Rabaey. *Digital Integrated Circuits*. Prentice Hall, 1996

[RSV97] J.A. Rowson, A. Sangiovanni-Vincentelli. "Interface-based design." *Proceedings of the 34th Design Automation Conference*, 1997. Pages: 178-183.

[SGR99] K. Stevens, R. Ginosar, and S. Rotem. "Relative timing." *Proceedings of. International Symposium on Advanced Research in Asynchronous Circuits and Systems ASYNC'99*, pp. 208-218, 1999.

[Si83] J. Sifakis. "Property preserving homomorphisms of transition systems." In E. Clarke and D. Kozen, editors, Proceedings of the 4th Workshop on Logics of Programs, Pittsburgh, U.S.A., June 1983.

[SS86] R. F. Sproull and I. E. Sutherland. *Asynchronous Systems.* Sproull and Associates, 1986.

[Su89] I. E. Sutherland. "Micropipelines." Communications of the ACM, 32(6) pp:720-738, June 1989.

[YD96] K.Y. Yun and R.P. Donohue. "Pausible clocking: a first step toward heterogeneous systems," *Proceedings of Computer Design: VLSI in Computers and Processors ICCD '96*, pp. 118 -123, 1996.

[Yi] L. W. Ying. *Verification and Re-Design of Communication Interfaces with Heterogeneous Timing*. Mater thesis, McGill University, 2001