Interface Optimization for Concurrent Systems Under Timing Constraints

David Filo, David C. Ku, *Member, IEEE,* Claudionor N. Coelho, Jr., and Giovanni De Micheli, *Senior Member, IEEE*

***Abstract*— The scope of most high-level synthesis efforts to date has been at the level of a single behavioral model represented as a control/data-flow graph. The communication between concurrently executing processes and its requirements in terms of timing and resources have largely been neglected. This restriction limits the applicability of most existing approaches for complex system designs. This paper describes a methodology for the synthesis of *interfaces* in concurrent systems under detailed timing constraints. We model interprocess communication using blocking and nonblocking *messages.* We show how the relationship between messages over time can be abstracted as a constraint graph that can be extracted and used during synthesis. We describe a novel technique called *interface matching* that minimizes the interface cost by scheduling each process with respect to timing information of other processes communicating with it. By scheduling the *completion* of operations, some blocking communication can be converted to nonblocking while ensuring the communication remains valid. To further reduce hardware costs, we describe the synthesis of interfaces on *shared* physical media. We show how this sharing can be increased through rescheduling and serialization of the communication. In addition to systematically reducing the interface synchronization cost, this approach permits analysis on the timing consistency of interprocess communication.**

***Index Terms*—Communication synthesis, concurrent processes, synchronization, timing constraints.**

I. Introduction

Past efforts in high-level synthesis have focused primarily on the synthesis of a single process [1]–[4]. Under this assumption, hardware behavior can be represented as a control-flow and/or data-flow graph, and tasks such as scheduling and binding are defined with respect to operations within a single process. While this assumption is adequate for uniprocessor synthesis, it is less effective in synthesizing more complex circuits and systems that are modeled best as multiple concurrent and interacting processes.

There are many examples of designs consisting of multiple interacting processes. Consider, for instance, a graphics enhancement unit modeled by two processes: an edge detection process and an image enhancement process. The edge detection process takes as input a stream of data representing the incoming image and detects its edge boundaries. This boundary information is then passed to the second process to be used for enhancement of the image via shading or highlighting. Fig. 1 shows a block diagram of this design. In general, the timing with which the edge detection process produces results is unknown, and therefore blocking communication is needed to coordinate its operation with its receiving process. However, it is possible in some cases to take advantage of the timing characteristics of the receiving process to reduce the amount of handshaking, and vice versa. In this case the combination of the sender and receiver allows the communication between them to be matched together, such that the result is a less general but simpler interface. The ability to exploit timing behavior of other processes during synthesis is key to obtaining an efficient implementation.

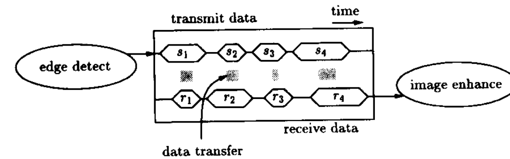


Fig. 1. Two interacting processes for a graphics application. The edge detection process is specified using all blocking communication for generality. Transformation to nonblocking communication can be synthesized when its communication with a second process (image enhancement) is matched.

Other examples include designs that interface under a given protocol (e.g., NuBus or EISA bus) and telecommunication applications. In these cases it is possible to specify generic interfaces without exact timing information using blocking communication. When such a process is synthesized in a system with a particular bus model, the timing information from the bus model is reflected back to the generic interface and the communication can be simplified by specializing it for the particular application.

Modeling a system as a collection of concurrently executing processes poses additional challenges to a synthesis system. In particular, synthesizing one process can in general alter the way it communicates with its environment. These changes in turn affect and constrain the synthesis of other processes in the system. The correctness of a design depends not only on the correctness of its data computations, but also on the timing and synchronization requirements that define when these results are communicated to and from the external environment. Of critical importance are the analysis and synthesis of the *interfaces* between the processes and the *protocol* governing their interaction, as well as their efficient implementation on shared physical media.

With few exceptions [5]–[9], existing techniques do not adequately address the synthesis of communication for concurrent

Manuscript received December 16, 1992; revised April 14, 1993. This work was supported by NSF/ARPA under Grant MIP-8719546, by DEC jointly with NSF, under a PYI Award program. C. N. Coelho, Jr. was supported by CNPq-Brazil under Contract 200212/90.7.

The authors are with the Center for Integrated Systems, Stanford University, Stanford, CA 94305.

IEEE Log Number 9210694.

1063–8210/93$03.00 © 1993 IEEE

systems. This paper presents a methodology for the analysis and synthesis of interfaces for time-constrained concurrent systems. Such systems are characterized by tightly interacting processes operating under strict timing and sequencing constraints. We abstract the interprocess communication using blocking, semiblocking, and nonblocking *messages.* These messages are transferred over abstract *communication channels,* which are mapped into a physical implementation (e.g. buses) by the synthesis process. This is in contrast to approaches where the communication is achieved structurally through the use of ports. We consider only point-to-point communication because of its determinism (i.e., each message operation communicates with exactly one other operation) and lack of arbitration, both important characteristics for time-constrained designs. We represent the timing and sequencing relationships between messages using a graph abstraction called a *message dependency graph.* This graph effectively captures the sequencing dependencies in the communication protocol and serves as the basis to analyze communication deadlock and cross-process timing requirements.

We present a novel technique called *interface matching* that minimizes the interface control logic and interprocess handshaking by scheduling each process using the communication patterns and timing behavior between concurrent processes. Our technique is guaranteed to yield the minimum amount of required explicit handshaking for a class of designs. In order to further reduce hardware costs, we describe the synthesis of communication on *shared* physical media. We show how this sharing can be increased through *rescheduling* and *serialization* of the communication.

The problem of reducing the amount of synchronization in concurrent processes has been studied in various forms. In the area of hardware synthesis, the approach described in [10] uses appropriately sized queues to decouple the sending and receiving processes. The use of queues allows processes to proceed with their execution without having to completely synchronize via blocking. This comes at the expense of increased implementation costs due to the added queues and increased control complexity. Our approach seeks to minimize the implementation cost by eliminating the need for queues by matching communication patterns between processes whenever possible.

In the area of concurrent software, the problem of minimizing synchronization has been explored at various levels [11]–[14]. The most relevant work to our problem is presented in [12] and addresses the problem of solving the synchronization problem for *barrier MIMD* machines [15]. In this problem, barriers are used to synchronize instruction streams running on concurrent processors. Although their model does not apply directly to our hardware synthesis problem, the goal of reducing the amount of synchronization through static resolution is the same as ours.

A block diagram of the proposed synthesis process is shown in Fig. 2. The first step is to schedule all of the processes independently using traditional scheduling techniques. This scheduling information is subsequently used by *interface matching,* which considers pairs of processes and attempts to simplify their communication. The result of matching may require the individual processes to be rescheduled, which can in turn lead to additional matching. Finally, *channel merging* is applied to reduce the number of physical channels. These steps will be described in detail, and it will be shown that this iterative optimization technique is guaranteed to terminate.

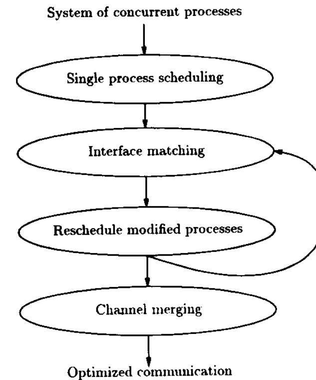


Fig. 2. Synthesis of communication.

The paper is organized as follows. Section II describes our model of hardware behavior and interprocess communication. The extraction of interfaces using message dependency graphs is described in Section III. Section IV describes the interface matching technique to reduce interprocess synchronization. The merging of communication channels is described in Section V. Finally, we conclude with experimental results and directions for further research.

II. Modeling Interprocess Communication

The choice of a hardware model largely impacts the scope and applicability of the synthesis algorithms. This work assumes that hardware is described using some generic hardware description language (HDL) that supports concurrency and interprocess communication. The HDL is compiled into a control/data-flow graph, which can then be optimized using the techniques described in this paper.

The *sequencing graph* model [16], [17] is used in order to leverage off of existing work. This model satisfies the necessary requirements and supports the explicit representation of detailed timing constraints and synchronization. We first give a brief overview of the sequencing graph model, then describe the extensions we have added to model interprocess communication. In this paper we restrict our focus to nonpipelined, synchronous designs.

A. Modeling a Single Process

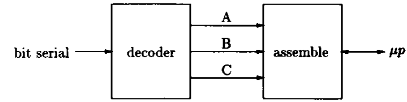
We model a single process as a polar, hierarchical *sequencing graph,* denoted by $G_s(V_s, E_s)$, where the vertices V_s represent operations to perform, and the edges E_s represent the sequencing dependencies among the operations. The sequencing graph is acyclic because loops are broken through the use of hierarchy. A process starts execution at the source vertex, executes each vertex according to the sequencing dependencies, and restarts execution upon completion of the sink vertex. The *execution delay* of a vertex v_i, denoted by

$\delta(v_i)$, can be *fixed* or *data-dependent.* The delay associated with a fixed delay operation depends solely on the nature of the operation, e.g., addition or register loading. In contrast, the time to execute a data-dependent delay operation may change for different input data sequences, e.g., waiting for the assertion of an external signal. We call the set of data-dependent delay vertices (including the source vertex) the *anchors of* $G_s(V_s, E_s)$, denoted by the set $A \subseteq V_s$.

Detailed minimum and maximum timing constraints can be specified between pairs of operations in the graph. In particular, consider two vertices v_i and v_j with start times $T(v_i)$ and $T(v_j)$, respectively. A *minimum* constraint $l_{ij} \geq 0$ between vertices v_i and v_j implies $T(v_j) \geq T(v_i) + l_{ij}$, and a *maximum* constraint $u_{ij} > 0$ implies $T(v_j) \leq T(v_i) + u_{ij}$. We derive a *constraint graph* $G(V, E)$ from a given sequencing graph $G_s(V_s, E_s)$ as the basis for timing analysis; the vertices are identical (i.e., $V = V_s$), but the edges E are now weighted. For a given edge $(v_i, v_j) \in E$, its edge weight w_{ij} corresponds to a timing constraint on the activation of the two operations v_i and v_j. Specifically, sequencing edges (E_s in the original sequencing graph) and the set of minimum timing constraints $\{l_{ij}\}$ are converted into *forward edges* $E_f \subseteq E$, and the set of maximum timing constraints $\{u_{ij}\}$ is converted into *backward edges* $E_b \subseteq E$. Forward (backward) edges have positive (negative) weights and represent minimum (maximum) timing requirements.

Example 1: Consider the network packet decoder example in Fig. 3. It consists of two processes: the main decoding process and a second process that assembles the data to present to the microprocessor. Fig. 3(b) shows the Verilog description for the main decoder process. The corresponding constraint graph for this process is shown in Fig. 4. The vertices are labeled according to their corresponding operation in the specification, and the source and sink vertices are labeled s and t, respectively. In addition to the sequencing constraints, both minimum and maximum timing constraints are shown in this example. Consider the *send* operation labeled c. The specification requires that the operations b and c must be sequential; therefore, a sequencing constraint exists from b to c. In addition, a maximum constraint between c and d requires c to begin execution no more than two cycles after d begins execution. The minimum constraint from c to e forces e to begin execution no less than $\delta(c) + 1$ cycles after c begins execution. In other words, e must begin execution no sooner than 1 cycle after the *completion* of c. □

Given the constraint graph for a process, we can synthesize an implementation that meets the required timing constraints, or detect if no such implementation exists, using relative scheduling [16], [17]. We review now some relevant background on relative scheduling. Recall from above that *anchors* are operations with data-dependent delays. For each vertex $v_i \in V$, the *anchor set* $A(v_i) \subseteq A$ consists of those anchors whose completion is needed to compute the start time of v_i (i.e. $T(v_i)$). A *schedule* for G is obtained by assigning an *offset* value $\sigma_a(v_i)$ to each anchor $a \in A(v_i)$ for all vertices $v_i \in V$. A valid (also called *wellposed*) schedule $\Omega(G) = \{\sigma_a(v_i), \forall v_i \in V\}$ is one that satisfies all timing constraints for all values of data-dependent delays. An anchor is *redundant* with respect to a vertex if removing it from consideration does not change the start time of the vertex, otherwise the anchor is called *irredundant.* The start time for an operation can be expressed solely in terms of its irredundant anchors [16], [17].



(a)

```
    // main decoder process
    always begin
a:  read_packet ();
    fork
      begin // preamble thread
b:      extract_preamble ();
c:      send (A, preamble)
      end
      begin // content thread
d:      extract_content ();
e:      send (B, content)
      end
      begin // parity thread
f:      extract_parity ();
g:      send (C, parity)
      end
    join
h:  cleanup ();
```

(b)

Fig. 3. A network packet decoder. (a) Block diagram. (b) Verilog description of the decoder process.

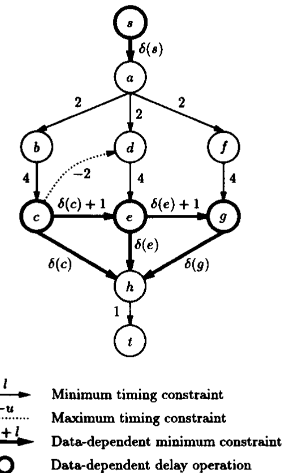


Fig. 4. Constraint graph for the decoder process example.

Example 2: Fig. 4 shows the constraint graph for the decoder process from Example 1 under a given profile of execution delays. The fixed delay operations have delays of $\delta(a) = 2, \delta(b) = \delta(d) = \delta(f) = 4$, and $\delta(h) = 1$. The bold vertices denote the anchors of the graph, and the bold edges represent forward edges weighted by a data-dependent delay. For example, the edge (c, e) has weight $\delta(c) + 1$, meaning e must wait at least 1 cycle after the *completion* of c. A valid schedule is given in Fig. 5, where offsets from each anchor are given. These offsets are simply the longest path, containing the delay of the anchor, between the anchor and the vertex. For example, the anchor set of vertex e is $A(e) = \{s, c\}$,

op	full schedule	irredundant
s		
a	$\delta(s)$	$\delta(s)$
b	$\delta(s)+2$	$\delta(s)+2$
c	$\delta(s)+6$	$\delta(s)+6$
d	$\delta(s)+4$	$\delta(s)+4$
e	$\delta(s)+8 \wedge \delta(c)+1$	$\delta(s)+8 \wedge \delta(c)+1$
f	$\delta(s)+2$	$\delta(s)+2$
g	$\delta(s)+9 \wedge \delta(c)+2 \wedge \delta(e)+1$	$\delta(e)+1$
h	$\delta(s)+9 \wedge \delta(c)+2 \wedge \delta(e)+1 \wedge \delta(g)$	$\delta(g)$

Fig. 5. Schedule for constraint graph of the decoder process example. The full schedule and irredundant schedule for each operation is shown.

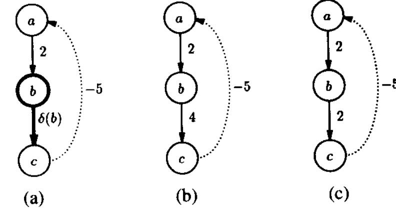


Fig. 6. Example of maximum constraint across a blocking operation. The original specification is shown in (a). (b) and (c) show possible results from interface matching when a nonblocking solution is found. Only (c) can be satisfied for all possible input traces.

and the corresponding longest paths (containing $\delta(s)$ and $\delta(c)$, respectively) have lengths of 8 and 1. Similarly, the start time of g depends on the completion of s, c and e. However, the offsets from s and c are not needed to calculate the start time of g; therefore, s and c are said to be *redundant* with respect to g, and the start time of g is simply $\delta(e) + 1$, as shown in the rightmost column. □

Since message operations represent points of interprocess communication, it is a natural point of reference for imposing timing constraints. Minimum timing constraints between message operations imply delaying the activation of messages and do not pose any problems. On the other hand, if there exist maximum timing constraints across blocking message operations, then the original specification may be overconstrained since a blocking operation has unbounded execution delay. With interface matching, it is possible to convert some blocking message operations to nonblocking operations by taking advantage of global communication patterns. We therefore modify the relative scheduling formulation in the following way.

Initially, any maximum timing constraints that are violated due to blocking operations are removed. After performing interface matching, these constraints are applied to the results and their consistency is checked. If blocking operations exist in the result, it is possible that some constraints will still be violated. Even if all operations are made nonblocking, the delays introduced by this process can result in constraint violations. It is shown later that under certain assumptions, if a solution exists, then interface matching will always find a solution.

Example 3: In Fig. 6(a) the maximum constraint from c to a is not valid because b is a blocking operation and its delay is unbounded, i.e., a positive cycle exists when $\delta(b) > 3$. In (b) the same constraint graph is shown after transforming b from blocking to nonblocking. Although the operation was made nonblocking, its delay ($\delta(b) = 4$) is too large and the graph is still overconstrained. Finally in (c), b is once again nonblocking, but its delay is within the bounds of the constraint. Therefore, a solution exists if b can be made nonblocking with $\delta(b) < 4$. □

B. Interface Between Processes

We abstract interprocess communication in terms of *messages* that are sent and received between processes over a set of abstract media called *channels*. Messages are assumed to be synchronous, taking one or more clock cycles to complete. Each message consists of a *send* operation and a *receive* operation. Returning to Fig. 3, the decoder process sends three messages $\{A, B, C\}$ containing the preamble, content, and parity information, respectively.

The communication can be *static* or *dynamic*. With static communication, a message has exactly one send operation and one receive operation associated with it. Multiple senders or receivers of the same message are not allowed. This means that send and receive operations can be statically matched to one another at synthesis time. In contrast, messages in dynamic communication are produced and consumed dynamically, often using queues to decouple the sending and receiving processes [10]. As stated before, we will restrict our focus to the synthesis of statically communicating processes.

A message operation can either be *blocking* or *nonblocking*. A blocking operation waits until its corresponding operation in another process is ready to execute. Once the correspondent is ready, the blocking operation executes with a fixed latency. A nonblocking operation assumes that its correspondent is ready and executes immediately without waiting. Nonblocking operations have a fixed latency, while blocking operations have an data-dependent delay. A blocking operation requires a control acknowledge from its correspondent to signal when it is ready, while a nonblocking operation does not require such handshaking signals.

A message, made up of two message operations, can be either *blocking, semiblocking,* or *nonblocking.* These types of messages correspond to the cases where both, one, or none of the operations are blocking. The number of control signals needed for handshaking is two, one, and zero, respectively. Nonblocking messages in our context are *unbuffered,* i.e., they are implemented as reads and writes to external ports without the use of queues and handshaking control logic. Nonblocking messages are useful when the sender and receiver are *implicitly coordinated.* As mentioned above, queues can be used to decouple the sender and receiver processes. This makes all messages nonblocking; however, queues introduce added cost to the implementation. Our goal is to eliminate the need for such queues through interface matching. If message operations remain blocking after matching, then queueing can be used in a complementary fashion.

When two processes are communicating via blocking message operations, it is possible that the process will *deadlock* [18], [19]. This happens if both processes are waiting for each other to execute some message operation that will in fact not execute until the currently stalled operation completes. An

obvious solution to avoid this type of deadlock is to make all message operations nonblocking, so that a process never has to stall while waiting for its corresponding operation in another process to execute. However, for unbuffered messages this implies the possibility of the receiver incorrectly sampling data before or after the sender is ready.

To ensure that data is properly communicated between the processes, we define a communication to be **valid** if two conditions are satisfied: (1) it is free of deadlocks, and (2) the send and receive operations of a message are coincident during the transfer of data for every message. Otherwise, it is an **invalid** communication. In other words, all messages that are transmitted are properly received.

The objective of interface synthesis is twofold: to analyze the communication for deadlocks and timing constraint violations, and to exploit the degrees of freedom in timing constraints to reduce synchronization and implementation costs while still ensuring valid communication. Referring back to Fig. 2, the interface matching step reduces the hardware costs by converting from blocking to nonblocking operations wherever possible. In this conversion, operations are transformed from having unbounded and unknown delays into known delays. Hence processes need to be rescheduled after such conversions to incorporate these known delays into the final schedule.

III. Extracting the Interface

We are now ready to formally define the *interface* between two processes. An interface describes two types of information: the *causal dependencies* between messages indicating whether executing one message requires the completion of other messages, and the minimum or maximum *timing relationships* that must be satisfied between the messages. Obviously, any timing relationship must be compatible with the causal dependencies. Intuitively, composing two processes consists of making sure the causal dependencies are mutually compatible, as well as propagating the timing relationships between the processes.

A. Message Dependency Graph

Given a process and its constraint graph $G(V, E)$, the set of *message operations* $M = \{m_1, m_2, \cdots, m_k\} \subseteq V$ in G represents the points at which the process interacts with its environment. We assume that all message operations have an unknown completion time (i.e., blocking), which implies they are also anchors in G, i.e., $M \subset A$. Each message is composed of exactly one *send* operation in the sending process and one *receive* operation in the receiving process.

For a process, represented by $G(V, E)$, communicating with other processes via message operations M, we define its *message dependency graph* as follows.

Definition 1: The **message dependency graph** of a process $G(V, E)$ with respect to a set of message operations $M \subseteq V$, denoted by $G_m(V_m, E_m)$, is a subgraph of G. The vertices consist of the message operations, i.e., $V_m = M$, and a directed edge $(v_i, v_j) \in E_m$ exists if $v_i \in A(v_j)$.

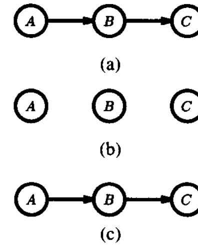


Fig. 7. Composing the message dependency graphs for the decoder and receiving processes. (a) Decoder. (b) Receiving process. (c) Composed message-dependency graph.

In other words, G_m captures the sequencing dependencies between message operations. Note that G_m is not necessarily connected. Fig. 7(a) shows the message dependency graph for the decoder example. It is easy to show that if the original graph G is valid, then the message dependency graph is acyclic.

Since G_m captures the causal relationships between message operations within a process, any valid communication between two processes must be *compatible* with respect to the causal relationships in the individual processes. To formalize this notion, consider two processes G_1 and G_2 communicating over a set of message operations M, with message dependency graphs G_{m1} and G_{m2}, respectively. We define the *composition* of G_{m1} and G_{m2} as follows.

Definition 2: The **composition** of two message dependency graphs G_{m1} and G_{m2} is a graph G_{m12}. The vertex set $V_{m12} = V_{m1} \cap V_{m2}$ consists of the common messages of the two processes. An edge (v_i, v_j) exists in G_{m12} provided there exists a path from v_i to v_j in either G_{m1} or G_{m2}.

The composition is a graph of the sequencing dependencies between common messages of two processes. As discussed previously, it is possible that the original specification results in communication deadlock. The message dependency graphs will be used to check for this condition. This initial analysis is necessary because later synthesis procedures assume that the communication is valid.

Theorem 1: Consider two processes P_1 and P_2 and their corresponding message dependency graphs G_{m1} and G_{m2}. If the composed message dependency graph G_{m12} contains a cycle, then the communication between the process is invalid.

Proof: Consider two vertices v_i and v_j on the cycle, which correspond to the two messages m_i and m_j. Let the associated send operations be denoted by s_i and s_j with the receive operations being r_i and r_j. For valid communication to exist s_i must be coincident with r_i, while s_j and r_j must likewise be coincident. For the sake of contradiction, assume that message m_i is valid. This means that s_i and r_i are coincident at some time t. The cycle in the graph implies there is a sequencing dependency from v_i to v_j. This means that one of the operations associated with m_j cannot execute until sometime later than t. Furthermore, there exists a dependency from v_j to v_i. This implies that the other message operation associted with m_j must complete execution sometime earlier than t. Clearly, message m_j is not valid since it is not possible for its message operations to be coincident in time. By

contradiction, a cycle in the dependency graph implies invalid communication. □

If the composed graph is cycle-free, then we say the communication is *consistent*. We state the following theorem:

Theorem 2: A consistent communication can always be made valid by making all message operations blocking.

Proof: By the definition of blocking communication it is clear that the send and receive operation are coincident. It remains to be shown that the communication is deadlock free. For the sake of contradiction, assume that two messages m_1 and m_2 are in deadlock. This means that in one process P_a, message m_1 is active and must complete before executing m_2, and in another process P_b, message m_2 is active and must complete before executing m_1. Therefore, in process P_a a causal path exists from m_1 to m_2, and in P_b a causal path exists from m_2 to m_1. By definition of the composed message dependency graph there exists a cycle containing messages m_1 and m_2. However, the composed message dependency graph is acyclic. So by contradiction we conclude that the communication is deadlock free. □

Example 4: Fig. 7 illustrates the composition of the decoder example (from Fig. 4), which sends three messages A, B, C, with a receiving process. Redundancies have been removed in the composed graph. The receiving process, which has not been shown, has no dependencies between the messages. This simply means that it is capable of receiving the messages in any order. In this case, the communication between the processes is consistent because there are no cycles in the composed graph. □

B. Incorporating Interface Timing Relationships

We have seen that consistency of the sequencing dependencies among message operations can be analyzed by composing message dependency graphs and checking for cycles. However, there are also timing relationships that are not represented in the message dependency graph abstraction, e.g., requiring a message operation to begin *at least* four cycles after the completion of another message. This timing information is needed by interface matching to ensure precise coordination of communication between the processes.

Transformation of a blocking operation into a nonblocking one is achieved through the use of timing information to schedule the *completion* time of the operation. In Fig. 2 the *start* times for all operations are computed in the initial scheduling step. The matching step attempts to compute the *as soon as possible* completion times so that nonblocking operations can be used in place of the original blocking ones. This implies that a schedule for the message dependency graph of a process is needed.

For the purposes of scheduling we assume that all message operations have unknown delay (i.e., they are anchors). This assumption is made because we want to schedule message operations with respect to other ones, and in our scheduling formulation the scheduling is done relative to the anchors. In addition to the message operations, there are other *internal* anchors not visible to other processes, such as data-dependent loops. In general, the start time of an operation may depend on such internal anchors.

For a process P, the **interface schedule** $\Omega^I(G_m)$ of $G_m(V_m, E_m)$ is defined to be a subset of the entire process schedule $\Omega(P)$ by restricting the start time of the message operations V_m to include only offsets from other message operations. To account for other anchors not included in V_m, an operation is called *controllable* if its start time refers only to message operations $a \in V_m$; otherwise, it is called *uncontrollable*. This property is critical in determining whether a message operation needs to remain blocking.

Example 5: In Fig. 5 the schedule for each of the message operations is given. Only the irredundant portion shown in the third column needs to be considered when computing operation start times; the rest is redundant. From the schedule we see that operation c depends on $\delta(s), e$ depends on $\delta(s)$ and $\delta(c)$, and g depends on $\delta(e)$. Therefore, c and e are said to be uncontrollable because their start times depend on a non-message operation, i.e., s. In contrast, operation g is controllable. □

In order to compute the completion times for some message operation a, we need to combine the interface schedules $\Omega^I(G_{m1})$ and $\Omega^I(G_{m2})$ of the two processes that use message a to communicate. The **composed interface schedule** $\Omega^I(G_{m12})$ defines the *as soon as possible* completion times for the message operations and can be computed as follows. Let $A_{m1}(v)$ and $A_{m2}(v)$ be the anchor sets of a message operation $v \in V_{m12}$ in the interface schedules $\Omega^I(G_{m1})$ and $\Omega^I(G_{m2})$, respectively. The anchor set for v in the composed interface schedule is the union of the anchor sets:

$$A_{m12}(v) = A_{m1}(v) \cup A_{m2}(v).$$

Let $\sigma_a^{m1}(v)$ and $\sigma_a^{m2}(v)$ be the offsets of an operation v with respect to an anchor a in the individual interface schedules[1]. The composed offset is computed as the maximum of the individual offsets, i.e.,

$$\sigma_a^{m12}(v) = \begin{cases} \max(\sigma_a^{m1}(v), \sigma_a^{m2}(v)), & \text{if } a \in A_{m1}(v) \cap A_{m2}(v); \\ \sigma_a^{m1}(v), & \text{if } a \notin A_{m2}(v); \\ \sigma_a^{m2}(v), & \text{if } a \notin A_{m1}(v). \end{cases}$$

Example 6: Consider the example in Fig. 8. The top part of the figure shows the message dependency graphs for two processes P_1 and P_2 with three messages $\{A, B, C\}$ before and after composition. There are sequencing constraints from A and minimum timing constraints between B and C. The bottom part of the figure shows an execution scenario for processes P_1 and P_2 based on schedules that are consistent with the individual processes. For example, message C and P_1 is scheduled to execute three cycles after the completion of A. If messages B and C are nonblocking, then the communication is not valid because the operations associated with the two messages would not be coincident. If the messages are made blocking, then operation B in P_1 would *wait* one cycle until its corresponding operation in P_2 executes, and operation C in P_2 would wait 3 cycles to synchronize with its correspondent

[1] It is possible the offset is undefined if a is not in the anchor set of v in the individual schedules.

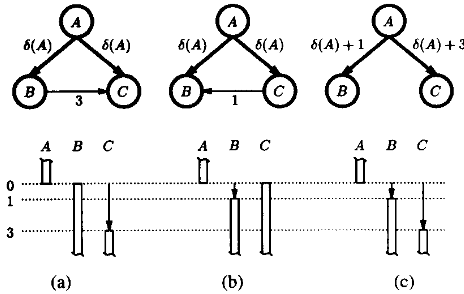


Fig. 8. Composing two message dependency graphs with timing constraints. (a) Process P_1. (b) Process P_2. (c) Composed schedule.

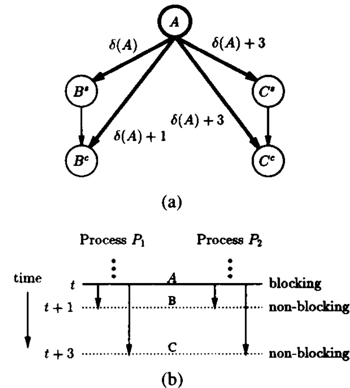


Fig. 9. Scheduling the decoder and receiver processes based on the composed interface schedule. Messages B and C can be made nonblocking without making the communication invalid. (a) Process P_1. (b) Matched schedule.

in $P1$. In the composed interface schedule (c), we see that if we schedule the *completion* of operations B and C *in both processes* to be 1 and 3 cycles after the completion of A, then they can be made nonblocking while still ensuring valid communication. In this case, the behavior is identical to the blocking case, but the hardware cost has been reduced. □

There are several important advantages to explicitly extracting and composing interfaces. First, it permits rigorous analysis of timing constraint consistency across process boundaries. Second, it enables each process to be synthesized individually, yet with all the requirements on its interactions with other processes fully represented as explicit timing constraints. Finally, it provides a formalism to manipulate and model interprocess interactions, e.g., we can now constrain the interface by directly applying sequencing and timing constraints on the external messages; these constraints can then be reflected to the individual processes for use during synthesis.

IV. Interface Matching

Given two processes, if their composed message graph is consistent, then Theorem 2 states that all messages can be made blocking to guarantee valid communication. However, it is often the case that the communication remains valid even if some messages are made nonblocking, as seen in Example 6.

Example 7: To further illustrate this point, consider process P_1 from Example 6. A possible schedule for P_1 is shown in Fig. 9(a). Both B and C have been broken into two vertices (e.g., B^s and B^c) to represent the start and completion of the operations. This allows the completion time to be scheduled under the constraints of the composed interface schedule shown in Fig. 8(c). A similar technique can be used in P_2 so that the resulting schedule is that of Fig. 9(b). A remains blocking while both B and C are made nonblocking. Although the operations start at different times, they complete at the same time, and a valid transfer takes place. This constitutes a significant savings in terms of synchronization logic. □

We formalize this observation by introducing the **interface matching** problem. Consider two processes P_1 and P_2 with common messages M and a corresponding composed message constraint graph G_{m12} that is acyclic. Let M be partitioned into the set of blocking M_{block} and nonblocking $M_{nonblock}$ messages. The interface matching problem is to minimize the number of blocking messages M_{block} while ensuring valid communication.

Intuitively, interface matching converts blocking messages into nonblocking ones by scheduling the *completion* time of a message operation when possible, as opposed to scheduling start times in conventional scheduling. This is because the completion of message operation implies the successful transfer of information between the sender and receiver. Therefore, interprocess communication can be viewed as a set of *time intervals,* where the start of the interval corresponds to the start of a message operation, and the end corresponds to its completion. Successful communication requires the intervals of the sender and receiver to always overlap at some point, for all message operations. The interface matching algorithm in the next section computes the *as soon as possible* point of overlap between the intervals. If a solution satisfying all constraints is found, then it is guaranteed to have minimum execution delay for all input data sequences.

Reducing the number of blocking messages leads to savings in two areas. First, blocking messages are implemented with a set of handshaking signals (e.g., request and acknowledge) to coordinate the data transfer between sender and receiver. Making a message nonblocking means these handshaking signals and the associated logic and ports can be removed. Second, a blocking message has a data-dependent execution delay. This can lead to larger controller cost because of the need to synthesize busy waits in both the sending and receiving processes. In contrast, no busy waits are necessary for nonblocking messages, which can result in a simpler control implementation [20].

A. Interface Matching Algorithm

The overall flow of the synthesis process was first outlined in Fig. 2. Many of the details were left out and are explained in more depth here. The algorithm for the interface matching is shown in Fig. 10. For simplicity, the low level details related to scheduling are not discussed here. Although we use relative scheduling, our formulation is independent of the scheduling technique, and other methods can easily be substituted.

Given a pair of scheduled communicating processes and a common set of messages, we first extract and compose their message dependency graphs. Although not shown in the

```
InterfaceMatch(P1, P2)
  forever
    // compute initial schedule
    Ω1 = Schedule(P1);
    Ω2 = Schedule(P2);

    // construct graphs
    Gm1 = ConstructMsgDependGraph(G1);
    Gm2 = ConstructMsgDependGraph(G2);
    Gm12 = ConstructComposedMsgDependGraph(Gm1, Gm2);

    // compute interface schedules
    Ω1^I = ConstructInterfaceSchedule(Gm1, Ω1);
    Ω2^I = ConstructInterfaceSchedule(Gm2, Ω2);

    // are there any controllable operations left?
    if all message operations m ∈ Vm1 ∩ Vm2 are uncontrollable
      break;

    // compute completion times for composed interface
    Ω12^I = ComputeCompletion(Gm12, Ω1, Ω2);

    foreach message m ∈ Vm12
      if m1 is controllable
        if m2 is blocking
          // make m2 non-blocking
          // set completion time for m2
          T(v2^c) = T(m);

      if m2 is controllable
        if m1 is blocking
          // make m1 non-blocking
          // set completion time for m1
          T(v1^c) = T(m);
```

Fig. 10. The interface matching algorithm.

algorithm, if the resulting composed graph is cyclic, then the communication is invalid and no solution is possible. Otherwise, for each process an interface schedule is derived from the process schedule. Based on these schedules, a schedule of completion times is constructed to form a composed interface schedule. If there are no controllable messages in either process, then there are no blocking operations that can be converted to nonblocking, and the algorithm completes.

For each message m in the composed message dependency graph, it has two message operations m_1 and m_2 from G_{m1} and G_{m2}, respectively. The corresponding vertices in the process constraint graphs G_1 and G_2 are denoted by v_1 and v_2. If the operation m_1 is controllable, then operation m_2 can be made nonblocking by scheduling its completion time $T(v_2^c)$. The completion time is set such that m_2 is guaranteed to complete after m_1 begins execution. This is possible because m_1 is controllable, and therefore its start time is known in P_2. If m_1 is blocking, the result is a semiblocking message where m_1 is ready first and waits for m_2, and if m_1 is nonblocking, the message is nonblocking where m_1 and m_2 complete at the same time. The same procedure is applied to m_2 and m_1 with their roles interchanged.

Example 8: An application of the algorithm is shown in Fig. 11. In this example we are only concerned with message c. In process P_1, the start time of c depends only on messages a and b; therefore, it is controllable. However, in process P_2, the start time of c depends on message a and an internal data-dependent loop d (denoted by the square vertex); therefore, it is uncontrollable. Operation c_2 (split into c_2^s and c_2^c) is made nonblocking in (b) by scheduling its completion (c_2^c), such that its completion is after the start of c_1 for all input traces. It is not possible to make c_1 nonblocking. □

After transforming all possible blocking messages, the entire process is started again by rescheduling and continuing from

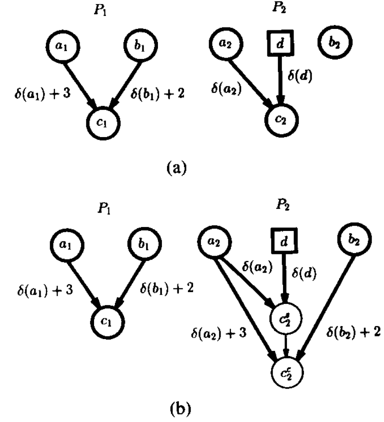


Fig. 11. Example of the matching algorithm, which results in a semiblocking message. The processes in (a) are before matching and (b) shows them after matching.

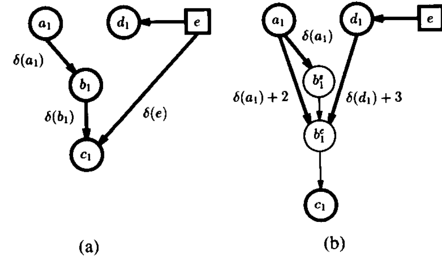


Fig. 12. Example where application of the matching algorithm introduces new controllable operations. Only one process is shown. Message operation c_1 is uncontrollable in (a), but after matching in (b), c_1 becomes controllable.

there. It is possible that the rescheduling will introduce new controllable operations that were previously uncontrollable. This happens when the rescheduled start time of a message operation no longer depends on an internal anchor.

Example 9: The need for iteration on the matching step is shown in Fig. 12. In this example we are concerned with messages b and c. In (a), b_1 is controllable while c_1 is uncontrollable because of the internal loop d. Suppose b_2 (in process P_2 not shown) is also controllable so that b_1 is made nonblocking. In this case, a new dependency from d_1 to b_1 is introduced in P_1. Because of the delay values, the start of message operation c_1 no longer depends on e; therefore, it becomes controllable. This cannot be known without rescheduling the graph. □

When there are no controllable operations remaining, the resulting constraint graph is rescheduled for a final time. This final scheduling is done without ignoring any of the maximum constraints. Remember that scheduling usually ignores maximum constraints across blocking operations because they may be converted to nonblocking ones. If one of these constraints is not satisfiable at the end, the original specification is overconstrained. It should be noted that in cases where some constraints remain unsatisfied, it might be possible to add serializations and constraint lengthening to introduce new controllable operations which potentially lead to a feasible result. However, these steps have the unwanted side effect of

reducing concurrency and increasing latency; therefore, they are not applied.

Since no changes in the existing constraints are made and no new ones are added, scheduling the completion time of an operation with this algorithm does not affect the cycle-per-cycle behavior of the resulting constraint graph. The algorithm simply determines the completion time of those blocking operations that can be made nonblocking. If the operation were to be left as blocking, it would complete at the same time as the transformed nonblocking operation. So the algorithm is guaranteed not to increase the latency of the design.

Under the restriction that the latency is not to be increased, this algorithm determines the maximum number of nonblocking operations. Any operation that remains blocking is the result of a uncontrollable operation. An uncontrollable operation can be made controllable only if a new sequencing edge is added or if the value of an existing constraint is increased. Modifying or adding such a constraint will lead to an increase in latency for some input traces. Therefore, the matching algorithm presented here computes the maximum number of nonblocking messages under the constraint that the latency is not increased.

The time complexity for one iteration of the matching algorithm over all processes is $O(|M||V|)$, where $|M|$ is represents the total number of messages in all processes and $|V|$ is the average number of vertices per constraint graph. The algorithm repeats itself if new controllable operations are formed. Typically only a few iterations are necessary to find all controllable operations. However, in the worse case $|M|$ iterations would be required, which results in an overall time complexity of $O(|M|^2|V|)$.

V. Channel Merging

Given an interface graph, the interface matching procedure reduces the number of blocking message operations as much as possible. This reduction in the number of blocking operations leads to lower communication and control costs. A further benefit is the potential for multiple separate communication channels (transferring data between sender and receiver) to be merged together and implemented on a shared physical medium. Merging is easier for nonblocking operations compared to blocking ones because they have fixed as opposed to unbounded delay. Until now, an assumption was made that each communication channel is implemented using dedicated control signals along with dedicated data lines. We now relax this assumption to allow the merging of channels.

There are varying degress to which channels can share the same physical hardware. In general, control signals can be shared separately from the data signals. Furthermore, depending on data widths, channels can be combined so that multiple channels can share the same physical channel at the *same time*. A more general scheme is to dynamically allocate the hardware to channels through the use of dynamic arbitration. Other variants are possible but we will consider only the static case where channels have the same size as the physical medium being mapped to. Furthermore, we assume that if two channels are shared, then both the data and control signals are shared.

Channel merging can be implemented at various stages of the synthesis flow. The most direct method, described in Section A, is to apply merging before scheduling by treating the physical channels as critical hardware resources and using serialization techniques [16] to share these resources. Alternatively, channel merging can be applied after scheduling has been performed, as described in Section B. In this case the results from scheduling are analyzed to determine where selective merging can take place. Finally, we describe in Section C *rescheduling* techniques to further increase the amount of merging that is possible.

A. Merging Before Scheduling

Merging before scheduling is the most direct method because communication channels are treated the same as other hardware resources, such as adders and ALUs. Therefore, traditional techniques used for resource sharing can be applied here with little or no modification. Provided the synthesis system can share critical resources under timing constraints, no special analysis is necessary to support this type of channel merging.

However, there is one important difference between communication channels and other hardware resources. Since we assume the initial message operations are blocking, the latency of these operations is not fixed. Therefore, given two candidate channels to merge, it is not enough to simply schedule the start time of the message operation of one channel before the start time of the other message operation. The reason being their completion times are unknown. A sequencing dependency between the operations must exist to guarantee that they are never active at the same time for all possible execution traces. Furthermore, this sequencing dependency must exist in both communicating processes for the channels to be merged while preserving valid communication.

An advantage of this method is that it can be used to solve the problem when the number of physical channels is constrained. Serialization can be used as a preprocessing step to ensure that the number of physical channels does not exceed the specified constraint. Techniques exist that finds a serialization under timing constraints, provided one exists [16]. The main disadvantage of merging before scheduling is that it requires operations to be serialized. The addition of sequencing dependencies to the constraint graph reduces the degree of concurrency, which may increase the latency of the final implementation. To avoid this increase, the technique presented in the next section can identify channel merging opportunities without affecting the latency.

B. Merging After Scheduling

To merge channels after scheduling, the first step is to analyze the scheduled results from the interface matching procedure. Based on the sequencing dependencies and schedules, it is possible to determine whether or not two channels can be merged. In this case merging has no effect on the synthesized result other than reducing the hardware cost. If changes in the circuit behavior (constrained by the specification of course)

are acceptable, further merging can be achieved by modifying the schedules. This will be discussed later.

After interface matching has been performed and the processes have been scheduled, channel merging is introduced by determining whether two given channels could possibly be active at the same time. If the message operations associated with two channels a and b are guaranteed to be mutually exclusive in time, both a and b can be merged together and implemented on the same physical channel. This analysis can be broken up into several different cases depending on whether or not message operations are blocking or nonblocking.

For the following cases, two messages a and b that communicate between two processes P_1 and P_2 are considered. The message operations associated with message a are denoted by a_1 and a_2 indicating the process to which they belong. Likewise b_1 and b_2 are the operations associated with message b. The anchor set of an operation is denoted by $A(x)$, where x is some message operation. As discussed previously the anchor set represents the set of operations having data-dependent delay upon which x is dependent for its activation. The two processes are analyzed separately, and later the results from both processes are combined to determine whether channels a and b can be merged.

The simplest case to analyze is when both a_1 and b_1 are blocking operations. As described in Section A, it is enough to check whether a sequencing dependency exists between a_1 and b_1. If such a dependency exists, the channels can be merged. Otherwise, they must remain separate.

If one of the two operations is nonblocking, say a_1, then the operation has been split into a start vertex a_1^s and a completion vertex a_1^c. Remember that the start of a_1^s represents the start of waiting, and a_1^c represents the actual transfer of data. In this case the delay of b_1 is not known (it is blocking). Therefore, a_1^c must complete before b_1 starts, for merging to be possible. Operation b_1 cannot be scheduled first because its completion time is unbounded. In order to guarantee that a_1^c completes before b_1, several conditions must hold. First, the anchor sets of the operations must have the relation $A(a_1^c) \subseteq A(b_1)$. This means that all data-dependent operations that affect the start time of b_1 must also affect a_1^c. Otherwise, it is possible that the completion of a_1^c will be delayed beyond the start of b_1, due to some *other* anchor. The second condition is that for each anchor in $A(a_1^c)$, its offset to the completion of a_1^c must be less than the offset to b_1. This ensures that for every possible execution trace, a_1^c always completes execution first. If these two conditions hold, the two operations can be merged.

Example 10: Fig. 13 illustrates the necessary conditions for the case of one blocking and one nonblocking operation. In (a) the anchor set relation is $A(b_1) \subset A(a_1^c)$. Since b_1 is the blocking operation, this violates the stated condition that $A(a_1^c) \subseteq A(b_1)$ must hold for merging to be possible. For example, if c and d begin at the same time (i.e., $T(c) = T(d)$) and the operation delays are $\delta(c) = 0, \delta(d) = 0$, and $\delta(b) = 3$, then a_1^c and b_1 would be active at the same time. In case (b) the anchor set relation is correct, i.e., $A(a_1^c) \subset A(b_1)$. However, contention would result in the case when $T(c) = T(d), \delta(c) = 0$, and $\delta(d) = 2$. Finally, in (c) both the anchor sets and offsets satisfy the conditions. For all possible execution traces a_1^c will start and complete before b_1 is started. Therefore, the two channels can be merged. □

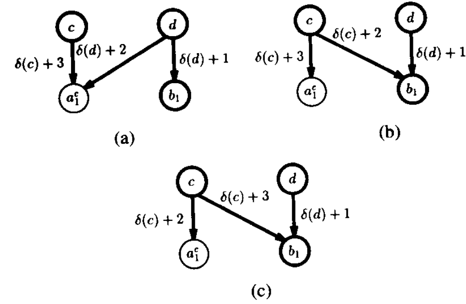


Fig. 13. Example of merge analysis when one operation b_1 is blocking and the other a_1^c is nonblocking. Bold vertices represent blocking messages. Operations cannot be merged in (a) and (b), but they can be merged in (c).

Finally, if both operations are nonblocking, the conditions to allow sharing are further relaxed. In this case the execution delay of both a_1^c and b_1^c are both fixed. The same analysis used in the previous case can be used here, with the difference that a_1 and b_1 are interchangeable in the above discussion. So for merging to be possible, it must be the case that $A(a_1^c) \subseteq A(b_1^c)$ or $A(b_1^c) \subseteq A(a_1^c)$. Furthermore, the offset from each anchor in $A(a_1^c)$ $(A(b_1^c))$ to the completion of a_1^c (b_1^c) must be less than the offset to the start of b_1^c (a_1^c).

Until this point, only a single process has been considered. However, for two channels to actually be merged, it must be the case that the operations can be merged in *both* processes. So for channels a and b to be merged, a_1 and b_1 must be merged in process P_1 as well as a_2 and b_2 in P_2.

Further analysis is also needed in order to merge more than two channels. Once all pairs of messages have been checked, a *merge compatibility graph* is formed where the vertices represent the messages, and an edge between two messages implies that the two can be merged. Two or more messages can be merged if and only if there exists a clique in the merge graph that contains the messages. Clique partitioning is performed on this merge graph to determine which messages are meged together. A physical channel is neded for each clique in the partition.

Example 11: In Fig. 14(a), the message constraint graph for some process P_1 is shown. For simplicity, assume the constraint graph for a second process P_2 is identical. Of the five messages in the graph, three of them $\{a, b, c\}$ are uncontrollable because they depend on a nonmessage anchor, while the other two $\{d, e\}$ are controllable. Therefore, the results from interface matching would yield three blocking and two nonblocking message operations in both processes. Analysis of this graph leads to the merge compatibility graph shown in (b) for both processes. Channel a can be merged with c and d, and b can be merged with d and e because of sequencing dependencies. Channels a and b cannot be merged because they are both blocking and no sequencing dependency exists between them. Channel c cannot be merged with either d or e because of incompatible anchor sets. The anchor sets of d and e have the relation that $A(e) \subseteq A(d)$, but the offset from b to d is less than to e. The minimum clique covering of the

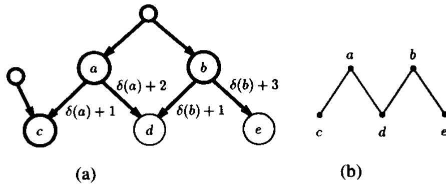


Fig. 14. Message constraint graph for processes P_1 and P_2 (same for both). Blank vertices represent nonmessage anchors. (a) Message-constraint graph. (b) Merge graph, which denotes merging possibilities between messages.

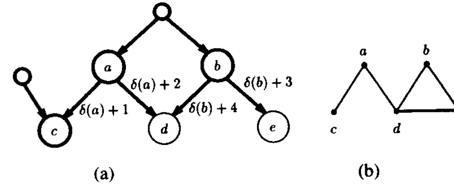


Fig. 15. Modified message constraint graph shown after rescheduling. (a) Message-constraint graph. (b) Degree of possible merging has increased.

merge graph results in three physical channels. For example, a possible merging would be $\{a, c\}, \{b, e\}, \{d\}$. □

C. Rescheduling Operations for Increased Merging

From the conditions for merging discussed in the previous section is should be clear that there is a better chance to merge nonblocking operations as compared to blocking operations. This means that the interface matching technique, by creating nonblocking operations, increases the ability to merge channels. Furthermore, the conditions also imply that additional steps can be taken to augment the merging process. Serializaing the messages by adding sequencing dependencies, modifying the anchor sets, and altering the scheduling offsets can be used to satisfy the conditions. The modification of anchor sets is accomplished by adding sequencing edges and/or modifying offsets, so we are left with these two techniques to improve the merging step.

There are several disadvantages to adding edges and increasing offsets. Although these techniques can be used in some cases to reduce control costs [20], they can only increase the latency and possibly increase the control costs. In the case of serialization, there are drawbacks in addition to the increased latency. Adding sequencing edges has the effect of reducing the degree of concurrency. Furthermore, these new edges can potentially change the anchor sets of operations in the graph. In some cases this is tolerable, but in our case modifying the anchor set of an operation can cause it to become uncontrollable. If this occurs, then the results from the matching step would be invalidated. Some nonblocking operations would have to be changed back to blocking. Therefore, there is no attempt to introduce new serialization after scheduling has been performed, and this technique should only be used before the interface matching procedure is applied.

So in order to obtain increased merging, we are left with modifying the existing schedule by increasing the value of constraints. It was shown that for two operations a and b to be merged the relation $\subseteq$ must hold between the anchor sets $A(a)$ and $A(b)$. If this relation does not exist, there is no need to reschedule the operations because they cannot be merged. Therefore the first step is to partition all the anchor sets into sets such that for each partition, an ordering among the anchor sets exists under the $\subseteq$ relation. For example, consider a case where we have the four message anchor sets $\{w, x\}, \{x, y\}, \{w, x, y\}$, and $\{w, x, z\}$. We can partition these into two sets $\{\{w, x\}, \{w, x, z\}\}$ and $\{\{x, y\}, \{w, x, y\}\}$ where the relation $\subseteq$ holds.

```
Reschedule(P)

  foreach message m_i in topological order
    Compute anchor set A_i;

  Construct a partition P of the anchor sets;
  // such that each p ∈ P has a complete ordering under ⊆

  Order each p ∈ P under ⊆;
  foreach (p ∈ P)
    foreach (a ∈ p)
      foreach (m|a ∈ A(m))
        increase path(a, m) if necessary;
```

Fig. 16. The rescheduling algorithm.

Example 12: The example in Fig. 15 is taken from Example 11. The only difference in this example is that rescheduling has been performed on operation d. We saw in the previous example that the anchor sets of d and e had the relation $A(e) \subseteq A(d)$. However, the schedules of these operations did not permit them to be merged. Operation d has been delayed by 3 cycles with respect to b to ensure that e will always execute before d. Doing this allows the two operations to be merged. The resulting merge graph is shown in (b). Now the minimum clique covering results in only two physical channels. The merged channels in this case are $\{a, c\}$ and $\{b, d, e\}$. □

The second step is to reschedule the operations based on how their anchor sets have been partitioned. The algorithm to partition the anchor sets and reschedule the operations is shown in Fig. 16. The algorithm is a heuristic that will find a new schedule such that maximum channel merging is achieved. However, it does not necessarily find the schedule with minimum latency. Due to the presence of unbounded delay operations, the meaning of minimum latency is not clear. The time complexity of the algorithm is $O(|A||M||V|^2)$, where $|A|$ is the number of anchors, $|M|$ is the number of messages, and $|V|$ is the number of overall vertices.

VI. Discussion and Limitations

If processes P_1 and P_2 contain any control-flow structure such as loops and conditionals, then their corresponding constraint graph representations are hierarchical in the sequencing graph model we are using. This causes difficulty in several areas. First, in our formulation timing constraints can only be specified between vertices of the same graph, which means that if we specify constraints between messages occurring in different graphs, they must be distributed across the hierarchy. Second, since we synthesize each graph in the hierarchy separately, it is necessary to partition the message links such that messages within each partition exist solely between two graphs in the hierarchy. Interface composition is then applied to each partition, in turn. Since the temporal relationship

between operations across the graph hierarchy is not directly captured, there is a possible loss of accuracy in the hierarchical extraction of timing relationships.

Until this point, we have ignored hierarchy and have only considered the communication between two distinct graphs in separate processes. Our methods can be applied to any two graphs at any level of hierarchy; however, none of the procedures work across hierarchy boundaries. We can deal with these problems in two ways. First, we can restrict as much of the communication as possible to a single graph in the hierarchy. Typically this would be the top level of the process. This has the limitation that the control reduction and channel merging techniques cannot be applied between communication events that occur at different levels of control hierarchy. A second approach is to reduce the hierarchy as much as possible by flattening the control structures. This could be done for most of the control structures except for loops which must be represented through hierarchy.

Although our techniques work on multiple processes, only simple point-to-point messages are supported. Messages with multiple senders or receivers (e.g., broadcasts) are not considered. More work is necessary to support other types of messaging and synchronization, which, in some cases, is highly desired. This would require the analysis of more than two processes simultaneously, which is currently not supported.

There are no restrictions on the relative repetition rates of processes for interface matching to be used. This is in contrast to some synthesis systems that assume all processes start and restart at the same time. This allows system designs that have a combination of processes iterating at varying rates. Communication has the effect of synchronizing such processes, but in general the processes remain synchronized for only a short time after completion of the communication. Interface matching takes advantage of this time when the processes are synchronized to simplify other communication. But once the processes cease to be in synchrony, no further optimizations can be achieved.

A limitation in the matching algorithm is that blocking operations are converted to nonblocking only if the latency of the result is not increased. So in the case when an increase is tolerable, the algorithm might not find the maximum number of nonblocking operations. This leads to a problem in the satisfaction of maximum timing constraints across blocking message operations. These types of constraints are allowed in the specification, but it is not always possible to find a solution satisfying all the constraints. If a solution is not found, it may be the case that the specification is simply overconstrained; however, it may also be the case that a possible solution was not found because it would result in increased latency. The solution to these problems is to allow the matching step to selectively allow increases in the latency when necessary.

VII. Results

Experimental results for three examples within the Olympus Synthesis system [21] are given in Table I. The table shows the number of blocking messages and ports before and after the optimization. The first column in the table gives the number of messages in the initial specification. The original messages are blocking and implemented using dedicated hardware; therefore the first three columns are the same.

TABLE I
Experimental Results of Applying the Interface Matching and Channel Merging Techniques

		Original		After	
Design	Messages	Blocking	Ports	Blocking	Ports
ECC encoder	16	16	16	1	1
ECC decoder	16	16	16	1	1
Elliptic filter	3	3	3	1	1
Receive	19	19	19	5	4
Decrypt	19	19	19	5	4

The first example consists of two processes in a system that models the transmission of digital data through a lossy serial line. The *encoder* process prepares the data and the associated parity information. This data is sent on a lossy line to a *decoder* process which uses the parity information to correct transmission errors if possible. In this example all of the communication is serialized in the original specification. Therefore, after optimization the first message remains blocking, while the rest have been converted to nonblocking. Furthermore, because of the serialization, all messages can be merged together using a single port.

The second example is the elliptic filter taken from [22], which has been partitioned into two processes using the *Vulcan* high-level partitioning tool. After partitioning, there are three communication messages used to transfer data between the partitions. These three messages can be simplified down to a single blocking message with two nonblocking messages all sharing a single port.

The final example consists of two processes from a decryption system. Due to control structures, both processes contain several levels of hierarchy in their constraint graphs. The message dependency graphs for these two processes are shown in Fig. 17. Graphs from the two processes that contain the same messages are paired together, and these pairs are processed separately. The receive process is responsible for handling the low-level details of reading data from an incoming serial line, extracting header and data information, checking for and correcting transmission errors, and sending the data off to a decryption process to obtain the clear text. The *decrypt* process receives data and proceeds to decrypt it based on the header information. While processing the incoming data, the clear text is sent back as it becomes available. Finally, some trailer information is sent back to the receive process. Due to the concurrency in this example, the final circuit still has multiple blocking messages.

VIII. Conclusion and Future Work

In this paper, we described an approach to the analysis and synthesis of interfaces for time-constrained concurrent systems. We proposed an explicit representation of the interface between processes in terms of *message dependency*

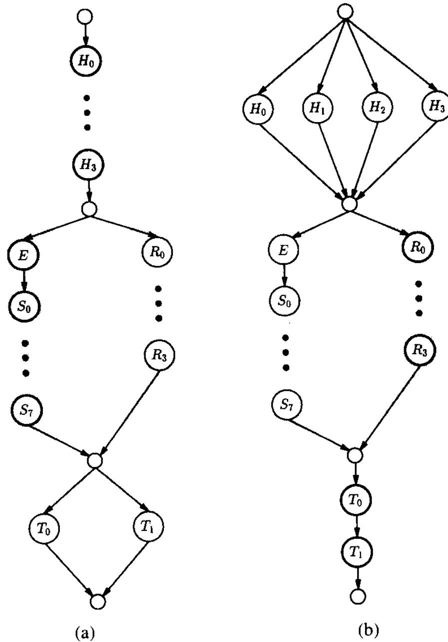


Fig. 17. Message sequencing graphs for decryption system. Levels of hierarchy are represented by shading. Normal vertices represent *receive* operations, and bold vertices represent *send* operations. (a) Receive process. (b) Decrypt process.

graphs. We described the *interface matching* technique to minimize the number of required blocking messages that is needed for valid, deadlock-free communication under detailed timing constraints. A method for sharing physical channels among multiple communication channels in order to reduce the communication hardware between processes was presented.

We are working to extend the formulation to better support hierarchy in the model. Currently, it is necessary to partition the messages such that messages within each partition originate from a single graph in the hierarchy and terminate in a single graph in another hierarchy. For many time critical designs where the control-flow structure of the sending and receiving processes is similar (to minimize the effect of control delays), this assumption is not a severe limitation. For other designs, there is potential loss of accuracy in extracting the timing requirements because the relationship across hierarchy may be lost. A solution is increase the scope of analysis by transforming the description to reduce the number of partitions, e.g., flattening or restructuring the control-flow. Another approach is to extend the formalism by using automata to describe the time progression of message operations on channels. These issues are currently under investigation.

REFERENCES

[1] M. McFarland, A. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proc. IEEE*, vol. 78, no. 2, pp. 301–318, Feb. 1990.

[2] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-level Synthesis: Introduction to Chip and System Design*. Norwell, MA: Kluwer, 1992.

[3] D. Thomas, E. Lagnese, R. Walker, J. Nestor, J. Rajan, and R. Blackburn, *Algorithmic and Register-Transfer Level: The System Architect's Workbench*. Norwell, MA: Kluwer, 1990.

[4] E. D. Lagnese and D. E. Thomas, "Architectural partitioning for system level synthesis of integrated circuits," *IEEE Trans. CAD/ICAS*, vol. 10, pp. 847–860, July 1991.

[5] J. Nestor and D. Thomas, "Behavioral synthesis with interfaces," in *Proc. Des. Automation Conf.*, June 1986, pp. 112–115.

[6] G. Borriello and R. Katz, "Synthesis and optimization of interface transducer logic," in *Proc. Int. Conf. Computer-Aided Design* (Santa Clara, CA), Nov. 1987, pp. 56–60.

[7] C. H. Gebotys, "Optimal synthesis of multichip architectures," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1992, pp. 238–241.

[8] Y.-H. Hung and A. C. Parker, "High-level synthesis with pin constraints for multiple-chip designs," in *Proc. Design Automation Conf.*, June 1992, pp. 231–234.

[9] S. Hayati, A. Parker, and J. Granacki, "Representation of control and timing behavior with applications to interface synthesis," in *Proc. Int. Conf. Comput. Design*, Oct. 1988, pp. 382–387.

[10] T. Amon and G. Borriello, "Sizing synchronization queues: A case study in higher level synthesis," in *Proc. 28th Design Automation Conf.*, June 1991.

[11] C. M. McNamee and R. A. Olsson, "Transformations for optimizing interprocess communication and synchronization mechanisms," *Int. J. Parallel Programming*, vol. 19, no. 5, pp. 357–387, Oct. 1990.

[12] H. G. Dietz, A. Zaafrani, and M. T. O'Keefe, "Static scheduling for barrier MIMD architectures," *J. Supercomputing*, vol. 5, no. 4, pp. 263–289, 1992.

[13] S. K. Tripathi and V. Nirkhe, "Pre-scheduling for synchronization in hard read-time systems," in *Operating Systems of the 90s and Beyond*, A. Karshmer and J. Nehmer, Eds. New York: Springer-Verlag, 1991, pp. 102–108.

[14] P. L. Shaffer, "Minimization of interprocessor synchronization in multiprocessors with shared and private memory," in *Int. Conf. Parallel Processing*, vol. 3 (St. Charles, Illinois), pp. 138–142, Aug. 1992.

[15] H. G. Dietz, T. Schwederski, M. T. O'Keefe, and A. Zaafrani, "Extending static synchronization beyond VLIW," in *Proc. Supercomputing '89* (Reno, NV), Nov. 1989, pp. 416–425.

[16] D. Ku and G. De Micheli, *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Norwell, MA: Kluwer, 1992.

[17] D. C. Ku and G. De Micheli, "Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits," *IEEE Trans. CAD/ICAS*, vol. 11, pp. 696–718, June 1992.

[18] L. Y. Liu and R. K. Shyamasundar, "Static analysis of real-time distributed systems," *IEEE Trans. Software Eng.*, vol. 16, pp. 373–388, Apr. 1990.

[19] K. M. Chandy and J. Misra, "The drinking philosophers problem," *ACM Trans. Programming Languages and Systems*, vol. 6, no. 4, pp. 632–646, Oct. 1984.

[20] D. Filo, D. C. Ku, and G. De Micheli, "Optimizing the control-unit through the resynchronization of operations," *INTEGRATION, VLSI J.*, vol. 13, pp. 231–258, 1992.

[21] G. De Micheli, D. C. Ku, F. Mailhot, and T. Truong, "The Olympus Synthesis System for digital design," *IEEE Design and Test Magazine*, pp. 37–53, Oct. 1990.

[22] R. Gupta and G. De Micheli, "Vulcan—a system for high-level partitioning of synchronous digital circuits," Stanford University, Stanford, CA, CSL Tech. Rep. CSL-TR-91-471, Apr. 1991.

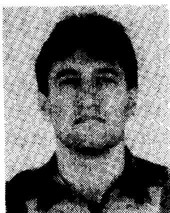
David Filo received the B.S.E. degree in computer engineering from Tulane University, New Orleans, LA, in 1988, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, in 1990. He is currently working toward the Ph.D. degree at Stanford.

His research interests include the synthesis of control and communication from a behavioral specification.

David C. Ku (S'87-M'91) received the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1987 and 1991, respectively. He received B.S. degrees in electrical engineering, *summa cum Laude*, and in computer science, *summa cum Laude*, both from the University of Utah,Salt Lake City, in 1986.

He currently leads the development of system-level design automation tools at Redwood Design Automation and continues as Research Associate at Stanford. He was a CIS/Signetics FMA Fellow in the Center for Integrated Systems at Stanford University during 1989–1991. He received the AT&T fellowship in 1986, the *Most Outstanding Senior in Electrical Engineering* award and the *Most Outstanding Junior in Electrical Engineering* from University of Utah, in 1986 and 1985, respectively.

Dr. Ku is on the program committee for ICCAD'93. He is a member of ACM.



Claudionor N. Coelho, Jr. was born in Niteroi, RJ-Brazil, in 1967. He received the B.S. degree in electrical engineering, Summa cum Laude, from the Universidade Federal de Minas Gerais, Brazil, in 1988. He is currently working toward the Ph.D. degree in the Department of Electrical Engineering, Stanford University, Stanford, CA.

His interests include high- and logic-level synthesis and verification.

Giovanni De Micheli (S'82-M'83-SM'89) received the Dr. Eng. degree, *summa cum laude*, in nuclear engineering from the Politecnico di Milano, Italy, in 1979, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1980 and 1983, respectively.

He is Associate Professor of Electrical Engineering and, by courtesy, of Computer Science, at Stanford University, Stanford, CA. From 1984 to 1986 he worked at the IBM T. J. Watson Research Center, Yorktown Heights, NY, where he was Project Leader of the Design Automation Workstation group. Previously, he held positions at the Department of Electronics of the Politecnico di Milano, Italy, and at Harris Semiconductor, Melbourne, FL. His research interests include several aspects of the computer-aided design of integrated circuits with particular emphasis on automated synthesis, optimization, and verification of VLSI circuits.

Dr. Micheli was granted a Presidential Young Investigator Award in 1988. He received the 1987 Best Paper Award for the best paper published in the IEEE TRANSACTIONS ON CAD/ICAS and two Best Paper Awards at the 20th Design Automation Conference, in 1983 and 1993. He is co-editor of *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation* (Martinus Nijhoff Publishers, 1987) and coauthor of *High-Level Synthesis of ASIC's Under Timing and Synchronization Constraints* (Kluwer, 1992). He was also codirector of the Advanced Study Institute on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, under the sponsorship of NATO in 1986 and 1987. He is Associate Editor of the IEEE TRANSACTIONS ON VLSI SYSTEMS and of *Integration: The VLSI Journal.* He was Associate Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS from 1991 to 1992. He was technical and general chairman if the International Conference on Computer Design—ICCD, in 1988 and 1989, respectively. He served as a member of the technical committee of the ICCD, ICCAD, and DAC Conferences. He also served as a member of the executive committee of the New York Chapter of the IEEE Computer Society in 1985 and 1986.