Interface Optimization for Concurrent Systems under Timing Constraints using Interface Matching

David Ku Dave Filo Claudionor N. Coelho, Jr. Giovanni De Micheli

Center for Integrated Systems
Stanford University

1 Introduction

Past efforts in high-level synthesis have focused primarily on the synthesis of a single process [1]. 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 graph. Whereas this assumption is adequate for uni-processor synthesis, they are less effective in synthesizing more complex and realistic DSP or *application-specific* (ASIC) designs consisting of multiple concurrent and interacting processes.

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 channels and busses.

With few exceptions [2, 3, 4], existing techniques do not adequately address the synthesis of communication for concurrent 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 inter-process communication using blocking and non-blocking *messages*. 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., knows exactly who is receiving what message) and lack of arbitration, both important characteristics for time constrained designs.

We model the timing and sequencing relationships between messages using a graph abstraction called an *causality dependency graph*. This graph effectively captures the sequencing dependencies in the communication protocol and serves as the basis for rigorous analysis on the cross-process interfaces. We present a novel technique called *interface matching* that minimizes the required inter-process handshaking by scheduling each process using timing information of the modules communicating with it. Our technique is guaranteed to yield the minimum number of required explicit handshaking, for a class of designs.

2 Modeling Concurrency

The choice of a hardware model largely impacts the scope and applicability of the synthesis algorithms. The *sequencing graph* model proposed by [5] is an appropriate starting point because of its explicit representation for detailed timing constraints and synchronization, along with the availability of analysis and synthesis techniques for time-constrained designs. We first give a brief overview of the sequencing graph model, then describe the extensions we have added to model inter-process synchronization.

Modeling a single process. We model a single process as a hierarchical *sequencing graph* $G_s(V, E_s)$, where the vertices V_s represent operations to perform and, the edges E_s represent the sequencing dependencies among 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 $G(V, E)$, denoted by the set $A \subseteq V$.

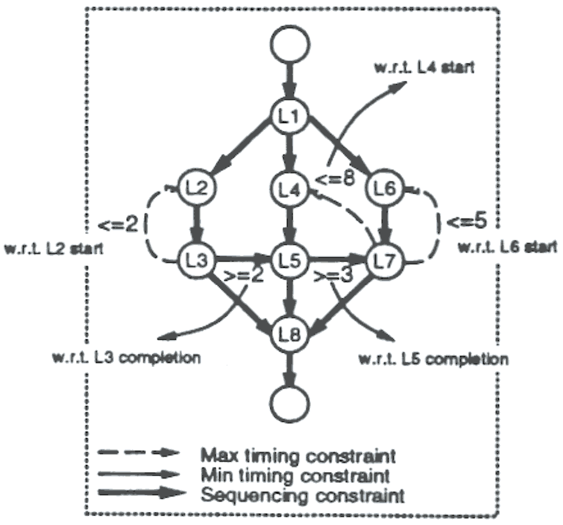
Detailed minimum and maximum timing constraints can be specified between pairs of operations in the graph. In particular, consider two vertices v_i and v_j with start times $T(v_i)$ and $T(v_j)$, respectively. A *minimum* constraint $l_{ij} \geq 0$ between vertices v_i and v_j implies $T(v_j) \geq T(v_i) + l_{ij}$, and a *maximum* constraint $u_{ij} \geq 0$ implies $T(v_j) \leq T(v_i) + u_{ij}$. We derive a *constraint graph* $G(V, E)$ from a given sequencing graph 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 arcs (in the original sequencing graph) and the set of minimum timing constraints are converted into *forward edges* $E_f \subseteq E$; the set of maximum timing constraints are converted into *backward edges* $E_b \subseteq E$. Forward (backward) edges have positive (negative) weights and represent minimum (maximum) timing requirements. An example of the sequencing graph for a network packet decoder under timing constraints is shown in Figure 1.

Interface between multiple processes. Given a constraint graph, we can synthesize an implementation that meets the required timing constraints, or detect if no such implementation exists, using relative scheduling [5]. We now extend the model to support inter-process communication.

We abstract inter-process communication in terms of *messages* that are sent and received between processes. Messages are assumed to be synchronous, taking one or more clock cycles to complete. Returning to Figure 1, the decoder process sends three messages $\{A, B, C\}$ containing the preamble, content, and parity information, respectively. The communication can be *static* or *dynamic*. In static communication, a message send action in one process is statically linked to a message receive action in another process. Because of the close relationship between processes, more can be done in matching their executions to optimize the communication. Unfortunately, this close relationship also results in several limitations, including the need to restrict the sending and receiving processes to have the same control-flow structure. In contrast, messages in dynamic communication are produced and consumed dynamically, often using queues to decouple the sending and receiving processes. As stated before, we will restrict our focus to the synthesis of processes with static communication. The extension to support dynamic

```
      {
L1:     packet = readPacket();
        {
L2:        preamble = extractPreamble(packet);
L3:        send(A, preamble)
        }
        parallel {
L4:        content = extractContext(packet);
L5:        send(B, content)
        }
        parallel {
L6:        parity = extractParity(packet);
L7:        send(C, parity)
        };
L8:     cleanup()
      }
```

(a) Behavioral description of decoder process



(b) Sequencing/constraint graph

Figure 1: A network packet decoder: (a) behavioral description, where ";" and "parallel" keywords denote serial and parallel execution, respectively, (b) corresponding constraint graph with 5 timing constraints (3 maximum and 2 minimum).

communication is currently under investigation.

A message can either be *blocking* or *non-blocking*. The distinction lies in that a blocking message requires explicit handshaking to establish connection before data is transferred, whereas no such handshaking is required for non-blocking messages. Therefore sending and receiving non-blocking messages have fixed delays, and blocking messages require data-dependent execution delays. Non-blocking 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. Non-blocking messages are useful when the sender and receiver are *implicitly coordinated*, i.e., the receiver is always ready to receive new messages.

With the assumption of single-sender and single-receiver under static communication, we make explicit the relationship between the message send action and the message receive action by an undirected edge between the respective graphs, called a *message event*. Figure 2 illustrates the three message events (shown as dashed lines) between the decoder process P_1 of Figure 1 and a slave process P_2. In some cases, the communication between two processes may stall because of circular dependencies in the execution order of the blocking message events. For example, consider two processes P_1 and P_2 communicating over two blocking message events A and B. In P_1, A executes before B, and in P_2, B executes before A. Upon executing A, P_1 will suspend execution waiting for P_2 to execute its A event. Simultaneously, P_2 will first execute event B, then wait for P_1. Neither processes can proceed and the communication is said to be in **deadlock**.

An obvious solution to avoid deadlock is to make all messages non-blocking, where a process is never halted to wait for incoming messages. For unbuffered messages, however, this implies the possibility of the receiver sampling when the sender is not ready. To ensure that data is properly communicated across the processes, we define a communication to be **valid** if two conditions are satisfied: (1) it is free of deadlocks, and (2) the sending and receiving processes are *synchronized*, i.e., they execute simultaneously at the point of every message event.

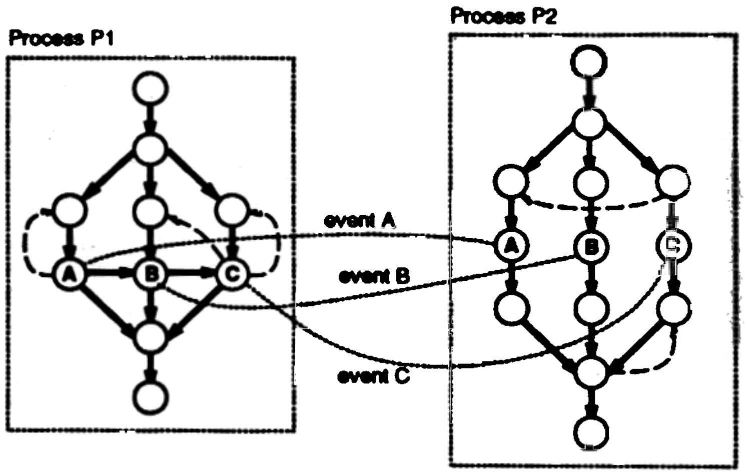


Figure 2: Modeling inter-process communication between processes P_1 and P_2 by three message events $\{A, B, C\}$, shown as dotted lines between the processes.

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 reduce synchronization costs while still ensuring valid communication.

3 Extracting the Interface

We are now ready to formally define the *interface* between two processes. An interface describes two types of information: the *causality dependencies* between events indicating whether executing one event relies on the completion of another event, and the *timing relationships* that must be satisfied between the events. Obviously, any timing relationship must be compatible with the causality dependencies. Intuitively, composing two processes consists of making sure the causality dependencies are mutually compatible, as well as propagating the timing relationships between the processes.

Before describing each part in detail, we first present the relevant background on constraint graphs. For each vertex $v_i \in V$, the *anchor set* $A(v_i)$ consists of the subset of anchors A whose completion v_i depends on before it can start execution. 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 is one that satisfies all timing constraints. We refer the interested reader to [5] for further details.

Figure 3(a) shows the constraint graph for our decoder example under a given set of execution delay information. The bold arcs represent forward edges weighted by a data-dependent delay, e.g., the edge $(L3, L5)$ has weight $\delta(L3) + 2$, meaning $L5$ must wait at least 2 cycles after the completion of $L3$. A valid schedule is given in (b), where the offsets are simply equal to the longest path length between the vertex and its anchor. Consider the schedule for $L5 = \{(6), L3(2)\}$; its anchor set consists of two anchors: the source vertex and $L3$, with offset values 6 and 2, respectively.

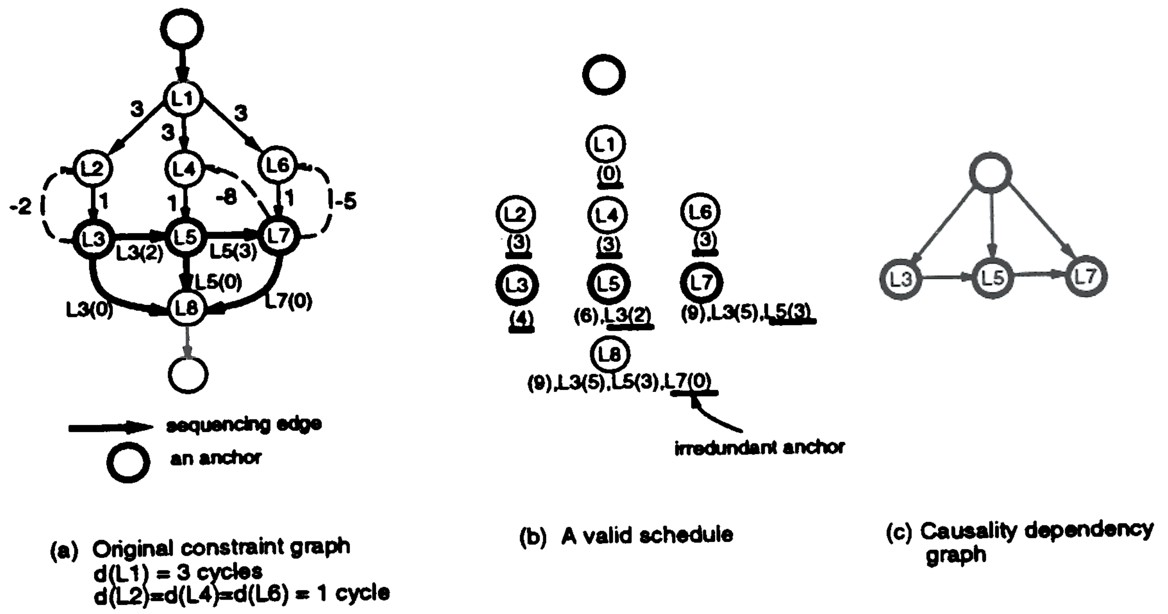


Figure 3: Illustrating the constraint graph for the decoder process example, (b) a valid schedule, where irredundant anchors are underlined, and (c) the corresponding causality dependency graph

3.1 Causality dependency graph

Given a process represented by a constraint graph $G(V, E)$, the set of message events $M = \{m_1, m_2, \ldots, m_k\} \subseteq V$ originating (or terminating) in G represents the points at which the process interacts with its environment. Initially, we assume all events to be blocking, which implies all events are also anchors in G. Each message event is associated with a *send* vertex in the sending process and a *receive* vertex in the receiving process. For a process represented by $G(V, E)$ communicating with other processes via message events M, we define its *causality dependency graph* as follows.

Definition 3.1 *The* **causality dependency graph** *of a process $G(V, E)$ with respect to a set of message events $M \subseteq V$, denoted by $G_c(V_c, E_c)$, is an induced subgraph of G. The vertices $V_c = \{v_0\} \cup M$ consist of the source vertex v_0 and the message events M. A directed arc $(v_i, v_j) \in E_c$ exists if $v_i \in A(v_j)$.*

In other words, G_c captures the sequencing (causal) dependencies between message events. Note that G_c is always connected, namely, all events depend on the source vertex. Figure 3(c) shows the causality graph for the decoder example. It is easy to show that if the original graph G is valid, then the causality dependency graph is acyclic.

Since G_c captures the causal relationships between message events 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 events M; let G_{c1} and G_{c2} be the respective causal dependency graphs. The vertex set for G_{c1} and G_{c2} is identical ($\{v_0\} \cup M$). We define the *composition* of G_{c1} and G_{c2} as follows.

Definition 3.2 *The* **composition** *of G_{c1} and G_{c2} is a graph $G_{c1 \times c2}$ with the same vertex set $V_{c1 \times c2} = V_{c1} = V_{c2}$, where an arc (v_i, v_j) exists if it exists in either G_{c1} or G_{c2}.*

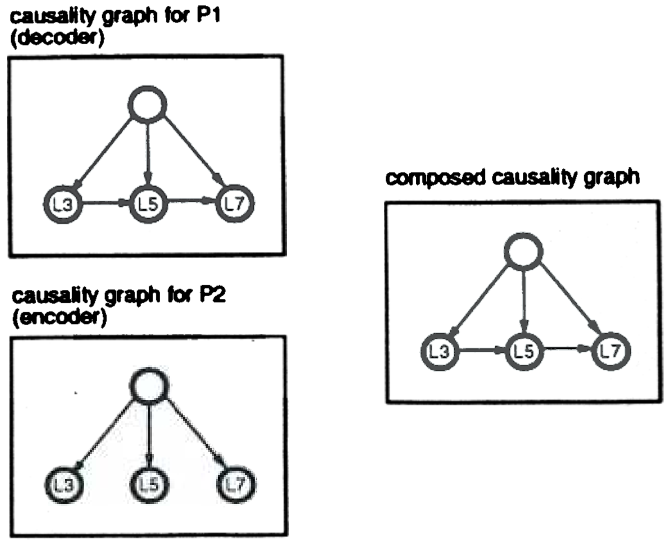


Figure 4: Composing the causality dependency graphs for the decoder and encoder processes.

Theorem 3.1 *If the composition causality graph $G_{c1\times c2}$ has a cycle, then the communication is invalid.*

Proof: Consider two vertices v_i and v_j on the cycle. The cycle means v_j depends on the *completion* of v_i, and v_i depends on the completion of v_j. If v_i and v_j are blocking message events, then the communication will deadlock because one process would be halted waiting for an event that can only occur if the process completes another operation, which depends on the first event occurring. If the events are non-blocking, then the two processes will not be synchronized after each message event, hence the communication is invalid. ||

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

Theorem 3.2 *A consistent communication can always be made valid by making all message events blocking.*

Proof: A consistent composed causality graph defines a partial order among message events. Any scheduling of the individual processes will necessarily be compatible with this partial order. Consider the case all message events are blocking. For a given event v, let $pred(v)$ be the set of immediate predecessors in the partial order, and let $succ(v)$ be the set of immediate successors. Consider the region of time between the completion of all events in $pred(v)$ and the start of all events in $succ(v)$. Compatible partial order means that there is no cyclic dependency between the $pred's$ and $succ's$ of the two processes. This means that the region of time can always overlap, which implies the event v can always be synchronized. Therefore, the communication is valid. ||

Figure 4 illustrates the composition of our decoder and encoder processes example. We will consider only consistent communication in the sequel, since otherwise the communication is invalid.

3.2 Incorporating interface timing relationships

Consistency of the sequencing dependencies among message events can be analyzed by composing causality dependency graphs and checking for cycles. However, there are also detailed timing relationships that are not represented in the causality graph abstraction. For this purpose, we *schedule* the causality graph G_c via relative scheduling [5]; the schedule defines for each event $v_i \in V_c$ offsets from a set of anchors satisfying the timing constraints in the original constraint graph G. We then perform redundancy removal to guarantee that, for a given schedule, a vertex depends on the *minimum* number of anchors. Since G is assumed to be wellposed, it is always possible to find the minimum and irredundant schedule [5]. In Figure 3(b), the irredundant schedule is underlined in bold.

All message events are anchors; the reverse, however, is not necessarily true. Examples of these *internal* anchors include data-dependent loops and conditionals. Therefore, the start time of an event may refer to delay information that is not externally visible. These internal data-dependent delays are important in determining whether a message event needs to be blocking (Section 4); however, their exact offset values are not as important as knowing that they exist. We define the **interface schedule** $\Omega_{ext}(G_c)$ of $G_c(V_c, E_c)$ as restricting the start time of an event to include only offsets from externally visible events in V_c. An event is called *controllable* if its start time refers to only externally visible events; otherwise, it is called *uncontrollable*.

Let us consider the effect of composing interface schedules $\Omega_{ext}(G_{c1})$ and $\Omega_{ext}(G_{c2})$. This basically means we would like to find a valid schedule for the composed causality graph $G_{c1\times c2}$. We assume $G_{c1\times c2}$ to be acyclic, since otherwise the communication is invalid. The *composed interface schedule* can be obtained as follows. Let $A_{c1}(v)$ and $A_{c2}(v)$ be the anchor sets of an event $v \in V_{c1\times c2}$ in the interface schedules $\Omega_{ext}(G_{c1})$ and $\Omega_{ext}(G_{c2})$, respectively. The anchor set for v in the composed interface schedule is the union of the anchor sets:

$$A_{c1\times c2}(v) = A_{c1} \cup A_{c2}$$

An event is *uncontrollable* in the composed schedule if it is uncontrollable in either of the interface schedules. Let $\sigma_a^{c1}(v)$ and $\sigma_a^{c2}(v)$ be the offset of an event v w.r.t. an anchor a in the individual interface schedules[1]. The composed offset is computed as the maximum of the individual offsets, e.g.,

$$\sigma_a(v) = \begin{array}{ll} \max\{\sigma_a^{c1}(v), \sigma_a^{c1}(v)\} & \text{if } a \in A_{c1}(v) \cap A_{c2}(v) \\ \sigma_a^{c1}(v) & \text{if } a \notin A_{c2}(v) \\ \sigma_a^{c2}(v) & \text{if } a \notin A_{c1}(v) \end{array}$$

Consider the example in Figure 5. The top part of the figure shows the causality dependency graphs before and after composition. The bottom part of the figure shows an execution scenario for $P1$ and $P2$, based on schedules that are consistent with with the individual processes. For example, for $P2$, event A is scheduled to execute one cycle after the source vertex. If the events A and B are non-blocking, then the communication is invalid because there would be no overlap between the event executions in the two processes. If the events are blocking, then event A in $P1$ would *wait* one cycle until its counterpart in $P2$ executes, and event B in $P2$ would wait 3 cycles to synchronize with $P1$. In the composed schedule, we see that if we schedule events B and A *in both processes* to be 3 cycles and 1 cycle after the source vertex, then they can be made non-blocking while still ensuring valid communication.

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 inter-process interactions, e.g., we can now constrain the interface by directly applying timing constraints on the external events; these constraints can then be reflected to the individual processes for use during synthesis.

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

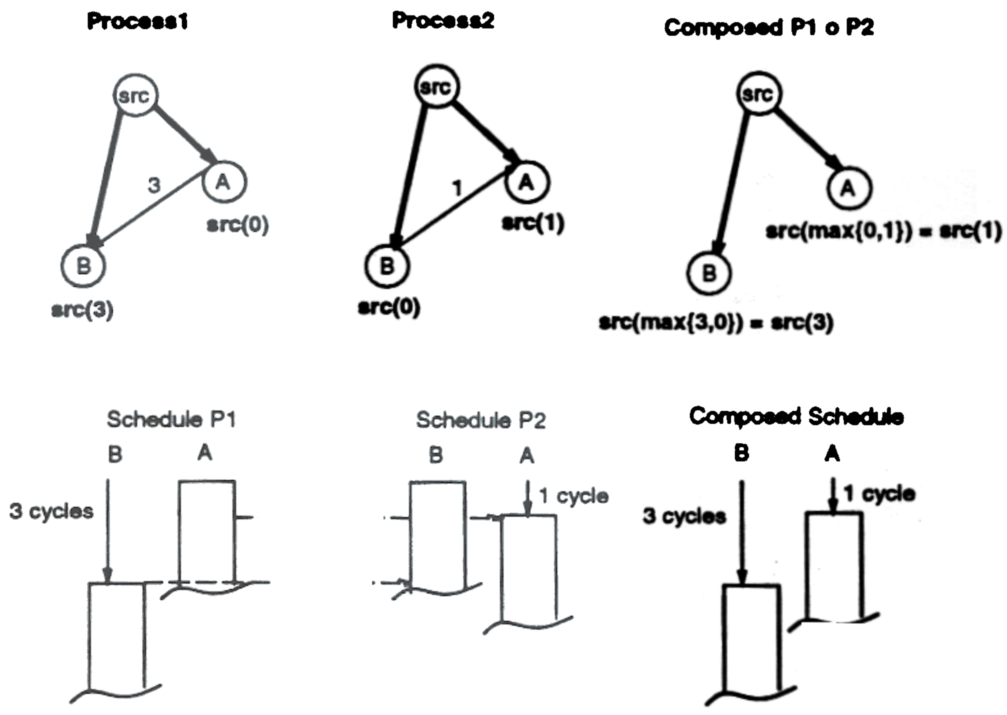


Figure 5: Composing two causality dependency graphs with respect to $\{A, B, C, D\}$.

Treatment of Hierarchy. If processes P_1 and P_2 contain control-flow, then their corresponding constraint graph representations are hierarchical. This causes difficulty in several areas. First, timing constraints can only be specified between vertices of the same graph, which means that if we specify constraints between message events occurring in different graphs, it 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 events 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 hierarchical extraction of timing relationships. In the case the control-flow structures of the sending and receiving processes are similar, this assumption is not a serious limitation. In general, however, it is necessary to restructure the control-flow by transformations, or relax the static communication assumption. Both strategies are currently under investigation.

4 Interface Matching

Given two processes, if their composed causality graph is consistent, then Theorem 3.2 states that we can always make all message events blocking to guarantee valid communication. However, it is often the case that the communication remains valid even if some events are made non-blocking instead. To illustrate this point, consider again the example of the decoder and encoder processes. From the composed causality graph, we see that the message events $\{L3, L5, L7\}$ are serially related. In particular, once the processes have been synchronized using event $L3$, the execution of subsequent events can be defined with respect to the completion of $L3$. Therefore, if

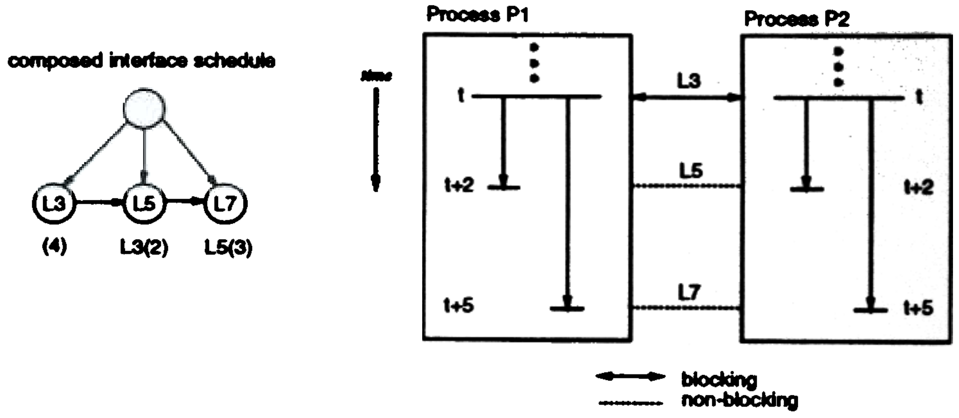


Figure 6: Scheduling the decoder and encoder processes based on the composed interface schedule. Events $\{L5, L7\}$ can be implemented as non-blocking without making the communication invalid.

we can schedule $L5$ and $L7$ to have the *same* offsets from the completion of $L3$ in *both* processes, then they can be implemented as non-blocking messages. One such schedule is illustrated in Figure 6; once processes P_1 and P_2 are synchronized by event $L3$ at time t, the remaining events in both processes can execute in lockstep. The resulting communication is still valid since the sending of a non-blocking message is executed simultaneously with the corresponding receiving action. This constitutes a significant saving 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 message events M and a corresponding cycle-free composed causality dependency graph $G_{c1 \times c2}$. Let M_{block} and $M_{nonblock}$ be the subset of blocking and non-blocking messages, respectively, where $M = M_{block} \cup M_{nonblock}$. The interface matching problem is stated as follows:

Minimize the number of blocking messages M_{block} while ensuring valid communication.

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 non-blocking means these handshaking signals and the associated logic and ports can be removed. Second, a blocking message event has 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 non-blocking messages, which can result in a simpler control implementation [6].

Before we present the details of the algorithm, we briefly describe the overall synthesis flow. Given a set of communicating processes and a common set of message events, we first extract and compose the causality dependency graphs. If the resulting composed graph is cyclic, then the communication is invalid and no solution is possible. Otherwise, find the minimum irredundant interface schedules for the individual processes and compose them together. The composed scheduling information is then the input to the interface matching algorithm, described next, to identify the minimum set of blocking messages. Once the blocking messages have been identified, the remaining messages are marked as non-blocking and have fixed execution delay of 1. The sequencing and timing constraints implied by the composed interface schedule are then reflected to the individual processes as external timing constraints, and each process is scheduled and synthesized accordingly. The resulting implementation is guaranteed to have valid, deadlock-free communication.

```
InterfaceMatch($E_M, G_\propto$) {
    foreach event $m_i$ in topological order {
        let $predSet = \{m_j | j = 0 \ldots i-1$ and $m_j \propto m_i\}$;
        let $predRoots = \bigcup_{p \in predSet} root(p)$;
        if $predRoots = \emptyset$ {
            $root(m_i) = \{m_i\}$;
        } else {
            let $\Gamma$ = all data-dependent delays in start time of $m_i$;
            if $\Gamma \subseteq predSet$ {
                $root(m_i) = predRoots$;
            } else {
                $root(m_i) = \{m_i\}$;
            }

    }
}
```

Figure 7: The interface matching algorithm.

4.1 Interface matching algorithm

The input to the interface matching algorithm is a composed causality graph $G_{c1 \times c2}$ and a corresponding consistent minimum, irredundant schedule. Each message event $m_i \in M$ has a corresponding label $root(m_i)$ that represents the set of blocking message events to which m_i depends upon for its activation. The label is initialized to the event itself: $root(m_i) = \{m_i\}$, implying all messages are initially blocking.

We define the *dominance* relation between events as follows.

Definition 4.1 *A message event m_i* **dominates** *another message event m_j, denoted by $m_i \propto m_j$ if two conditions hold: (1) m_j is controllable, and (2) the start time of m_j contains the offset from m_i.*

In other words, $m_i \propto m_j$ if the activation of m_j depends on a given offset from the completion of m_i. Condition (1) states that an uncontrollable event can never be dominated. Since the composed causality graph is acyclic, the dominance relation is acyclic and transitive. We call the graph induced by the dominance relation as the *dominance graph* $G_\propto$. The dominance graph for the example in Figure 6 is a chain, from $L3 \rightarrow L5 \rightarrow L7$.

Since $G_\propto$ is acyclic, it induces a topological ordering among the events. The interface matching algorithm visits each message event m_i according to its topological order, setting the corresponding label $root(m_i)$. Upon visiting all events, the set of blocking messages M_{block} is exactly equal to the union of all labels:

$$M_{block} \equiv \bigcup_{m_i \in M} root(m_i)$$

The algorithm is described in detail in Figure 7. Intuitively, it determines which messages must be blocking and partitions the rest of the messages into groups that are scheduled with respect to the blocking ones. Messages must be blocking if their start time depends on some internal delay that is not available to other processes.

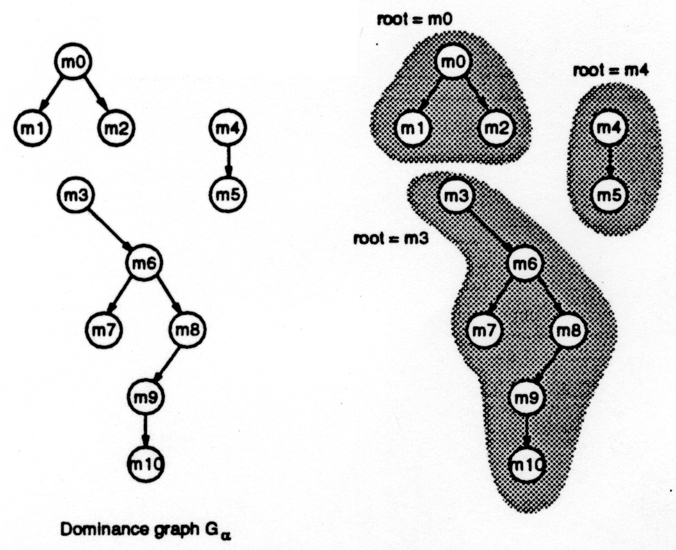


Figure 8: Example of the interface matching algorithm, showing the dominance graph $G_\propto$ and the resulting labeling of events. The transitive relationship in $G_\propto$ is omitted for simplicity.

We illustrate the algorithm on the example in Figure 8. The dominance graph $G_\propto$ is shown in (a) and the root labels for each event is shown in (b). The topological ordering of the traversal is reflected in the naming of the message events, starting from m_0 and ending at m_{10}. The result implies that we require only three blocking messages $\{m_0, m_3, m_6\}$ in the final implementation. Given the original causality dependencies, this algorithm produces the minimum number of blocking messages. However, by allowing additional causality dependencies through the use of serialization [6], the minimum set of blocking messages is potentially reduced and can be found with a modified algorithm.

5 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 *causality dependency 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. Preliminary experimental results within the Olympus Synthesis system [7] are encouraging. Table 1 shows the number of blocking messages and ports before and after the optimization.

We are working to extend the formulation to better support hierarchy in the model. Currently, it is necessary to partition the message events such that events 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

Design	*# Messages*	*Original*		*After*	
		# block	ports	# block	ports
ECC encoder	16	16	32	1	2
ECC decoder	16	16	32	1	2
packet xmit	3	3	6	1	2
packet recv	3	3	6	1	2

Table 1: Experimental results of applying interface matching technique.

is to extend the formalism by using automata to describe the time progression of message actions on channels. These issues are currently under investigation.

Acknowledgements

This research was sponsored by NSF/ARPA, under grant No. MIP 8719546, by AT&T and DEC jointly with NSF, under a PYI Award program. The third author was supported by CNPq-Brazil under contract 200212/90.7.

References

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

[2] J. Nestor, "Specification and synthesis of digital systems with interfaces," ph.d. dissertation, Carnegie-Mello University, Apr. 1987.

[3] G. Borriello and R. Katz, "Synthesis and optimization of interface transducer logic," in *Proceedings of th International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 56–60, Nov. 1987.

[4] G. Borriello, "Combining event and data-flow graphs in behavioral synthesis," in *Proceedings of the Interna-tional Conference on Computer-Aided Design*, (Santa Clara, CA), Nov. 1988.

[5] D. Ku and G. D. Micheli, *High Level Synthesis of ASICs Under Timing and Synchronization Constraints* Kluwer Academic Publishers, June 1992.

[6] D. C. Ku, D. Filo, and G. D. Micheli, "Control optimization based on resynchronization of operations," i *Proceedings of the Design Automation Conference*, June 1991.

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