An Approach to Interface Synthesis

Madsen, Jan; Hald, Bjarne

Published in:
Proceedings of the 8th International Symposium on System Synthesis

Link to article, DOI:
10.1109/ISSS.1995.520607

Publication date:
1995

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
An Approach to Interface Synthesis *

Jan Madsen and Bjarne Hald
Department of Computer Science
Technical University of Denmark, DK-2800 Lyngby, Denmark

Abstract
This paper presents a novel interface synthesis approach based on a one-sided interface description. Whereas most other approaches consider interface synthesis as optimizing a channel to existing client/server modules, we consider the interface synthesis as part of the client/server module synthesis (which may contain the re-use of existing modules). The interface synthesis approach describes the basic transformations needed to transform the server interface description into an interface description on the client side of the communication medium. The synthesis approach is illustrated through a point-to-point communication, but is applicable to synthesis of a multiple client/server environment. The interface description is based on a formalization of communication events.

1 Introduction
System level design may be viewed as the process of mapping a conceptual model into a physical structure of cooperating components. In this view a system is considered as a set of servers and clients that communicate via a communication medium.

In this paper we address the problem of automatically obtaining a customized implementation of the interface between client/server modules, termed interface synthesis. The main motivation is to adapt the interface during system implementation, rather than having a fixed communication architecture as is the case in most hardware/software codeign approaches, e.g., [2, 5, 6] which are using memory mapped I/O.

The simplest system consists of a single client invoking one operation from a server, i.e., a point-to-point communication. This corresponds to the traditional view of hardware/software codesign approaches, where we initially have an application which can not fulfill some given timing requirements. In order to meet these timing requirements, a subtask suitable for speedup is identified and moved to another module (hardware or software) capable of achieving the required speedup, i.e., the original application becomes a client which has to invoke the server in order to complete its computation.

We present a general interface model and an approach to interface synthesis which allows for communication optimization during client/server synthesis.

*This research has been sponsored by the Danish Technical Research Council under the "Codesign" programme.

2 Problem Formulation
Our synthesis approach is described in figure 1. The figure outlines the different steps involved in synthesizing a point-to-point communication. After system partitioning, which also selects the high-level communication protocol between the client and server, we first synthesize the server as this is the task to be speeded up. When synthesizing the server we may use traditional hardware synthesis in order to create a new implementation or we may re-use an existing module. Having synthesized the server, the next step is to synthesize the channel from the selected media. The media selection may be guided by the achieved server speed-up and the system timing requirements, or it may be guided from the system partitioning. Finally, we may synthesize the client to complete our system.

This approach is different from the ones in [3, 7] where both client and server are assumed implemented before interface synthesis. In [3] both modules may, however, be rescheduled to fulfill timing requirements. In [7] the focus is to optimize the channel utilization by interleaving different point-to-point communications on the same medium, re...
The PFG prescribe that the client has to send the values a multiplier (FMULT) with data dependent execution time. The protocol selected during the partitioning phase prescribes how and when to provide operands and pick up results from the server in order to execute a particular server operation. From the protocol description we may extract the interface of the server, i.e., a one-sided interface describing how the client has to interact with the server in order to perform the server operation. The interface synthesis is defined in terms of this interface description. Two interface synthesis tasks may be identified;

1. **Server implementation** which transforms the abstract interface description into an implementation defining the exact sequence and timing of transfers.

2. **Channel synthesis** which maps the server interface to the client-end of the channel, i.e., describing how to transfer data and control to the server using the interface of the channel. This task is also referred to as channel mapping [9].

In this paper we are focusing on channel synthesis, i.e., how to obtain a physical implementation of the channel. Thus, we assume that the server has already been implemented and that the medium (or set of media) has already been selected.

### 3 Interface Protocol Representation

The interface protocol to a module is specified as a Protocol Flow Graph (PFG) which prescribes how and when to provide input and receive output from a module. A PFG may represent both abstract and concrete interface protocols; An abstract interface protocol corresponds to an interface at the specification level, i.e., before module synthesis. The abstract PFG is extracted from the protocol description obtained during partitioning. A concrete interface protocol corresponds to an interface at the implementation level, i.e., after module synthesis (scheduling and allocation). A detailed description of the implementation of functional modules and the relation to PFGs may be found in [4].

Figure 2a illustrates the abstract PFG for a fixed point multiplier (FMULT) with data dependent execution time. The PFG prescribes that the client has to send the values a and b to the server, and when the flag d is raised the result c may be received by the client.

The interface protocol specified by a PFG may formally be described in terms of communication events. Throughout this paper we will use the notation based on communication events to describe our interface synthesis approach, and the graph based PFG notation to visualize various interface protocols.

#### 3.1 Interface Protocol Notation

A basic communication event is concerned with the transfer of a single value;

![Diagram](image_url)

**Figure 2:** The Protocol Flow Graph for a fixed point multiplier: a) Abstract view, b) Implementation view.

**Definition 1** Let \( e \) be a basic event defined as:

\[
e \triangleq \preceq v \! \vdash v
\]

where \( v \) is a value to be transferred, and \( \preceq v \! \vdash v \) prescribes input and output values respectively.

As an interface protocol prescribes the order of value transfers, we define the following temporal relations between events;

**Definition 2** Let \( op \) denote a temporal relation between two events, i.e., \( e_1 \! \preceq op \! \vdash e_2 \). There are four relations; two serial and two parallel:

\[
\begin{align*}
\text{Serial} (\preceq_s) & : e_1 \! \preceq_s e_2 \quad e_1 \text{ and } e_2 \text{ are executed sequentially in any order.} \\
\text{Serial} (\preceq_s) & : e_1 \! \vdash_s e_2 \quad e_1 \text{ and } e_2 \text{ are executed sequentially and with } n \text{ - 1 cycles in between.} \quad n = 1 \text{ is denoted } \vdash_s \text{ and indicates consecutive cycles. The case where any number of cycles may elapse in between the two events is denoted } \vdash_s^* \\
\text{Parallel} (\parallel) & : e_1 \! \parallel e_2 \quad e_1 \text{ and } e_2 \text{ may be executed in parallel or in any order.} \\
\text{Parallel} (\parallel) & : e_1 \! \vdash_s e_2 \quad e_1 \text{ and } e_2 \text{ has to be executed in parallel.}
\end{align*}
\]

In this context parallel means simultaneously within the same cycle, where a cycle is defined as the period between consecutive events on a synchronization signal, e.g., the rising edge of a synchronous clock. A set of value transfers which has to be transferred within a given fixed number of cycles, is denoted a timed event, i.e,

**Definition 3** A timed event \( t \) is an event defined as:

\[
t \triangleq \preceq v \! \vdash v | op \! \vdash e
\]

where,

\[
\text{op} \neq \vdash_s^*
\]

\( i.e., \text{an event in which no relation of type } \vdash_s^* \text{ occurs.} \)
Example 1: At the implementation view, a timed event relates values to be transferred to actual cycles. This means that in some cycles no values are to be transferred. Considering the example of a 4 cycle adder; in the first cycle the two operands $a$ and $b$ are transferred to the adder and in cycle 4 the result $c$ is transferred from the adder. This may result in the following timed event:

$$ t_{add} = (\langle a \| \langle ?b \rangle \rangle \langle ?c \rangle) $$

Synchronization is obtained through special synchronization events. A synchronization event blocks the communication until some condition becomes true. The condition is evaluated on values obtained through a set of value transfers. This scheme corresponds to a generalization of the implementation of handshaking as described in e.g., [7]. We define the synchronization event as:

**Definition 4** A synchronization event $w$ is an event which is repeated until some boolean expression $\text{expr}$ becomes true:

$$ w \triangleq (e : \text{expr})^+ $$

the $\text{expr}$ is always executed once.

the $\text{expr}$ is evaluated by the client on data obtained from $e$, i.e., a synchronization event implements a polling mechanism.

We can now give the complete definition of an event;

**Definition 5** Let $e$ be an event recursively defined as:

$$ e \triangleq \langle v \mid v \mid e \mid \langle e : \text{expr} \rangle^+ $$

If an event describes the complete interface protocol to a module, we will denote this event a PFG in order to relate the two notations, i.e., $\text{PFG} \equiv e$.

3.2 Server Interface

The interface to the server is described as a PFG. After server synthesis, detailed information about the sequence and timing of data and control transfers have been determined. This may be reflected in the PFG as illustrated in figure 2b. Each timed event is now associated with an instruction, which must be invoked in order to perform the actual data transfer. The implementation view of the PFG is an extended version of the protocol description used in AMICAL [8].

Example 2: As an example, the PFG of figure 2a, i.e., the specification view, may formally be expressed as:

$$ \text{PFG}_{\text{MULT}} = (\langle ?a \| ?b \rangle \langle \downarrow d : d = 1 \rangle^+ \langle \uparrow c \rangle) $$

and the PFG of figure 2b, i.e., the implementation view, as:

$$ \text{PFG}_{\text{MULT}} = (\langle ?\text{req} \| ?a \| ?b \rangle \langle \uparrow \text{req} \| \downarrow d : d = 1 \rangle^+ \langle \downarrow \text{ack} \rangle) $$

Notice the introduction of instruction invocations in the implementation view.

3.3 Medium Interface

The communication medium, $m$, takes care of the physical data transport. Examples of communication media are on-chip data busses, collections of wires or a VMEbus. On an abstract level each medium provides the possibility of sending and receiving data. On a lower level each medium specifies the protocol to send and receive data, e.g., 4-phase handshake or fixed-delay, and the data size to be transferred. In our representation each medium provides a READ (\sim!) and a WRITE (\sim?) operation for which the low level interface is described by a PFG; $\text{PFG}_{\text{MULT}}$ and $\text{PFG}_{\text{MULT}}$ respectively. Figure 3a shows the WRITE operation for a synchronous medium; the implementation view (\langle ?a \rangle) specifies that the next data value cannot be transferred until at least 3 cycles have elapsed. Figure 3b shows an example of an asynchronous medium; $\text{PFG}_{\text{MULT}} = (\langle ?\text{req} \| ?a \rangle \langle \uparrow \text{req} \| \downarrow \text{ack} \rangle)$ specifies a handshake protocol where the values $\text{req}$ and $a$ are send, and where the next data value cannot be transferred until $\text{ack}$ has been received.

4 Channel Synthesis

A channel is an adaptation of a medium (or set of media) to a client/server configuration. The channel is thus the outcome of interface synthesis. The need for a channel representation arises from the need to map the server PFG to the client-end of the medium. In this context the channel provides the necessary access operations in order for the client to be able to invoke an operation in the server.

The adaptation of a medium to a client/server configuration requires control logic and memory at the server-end of the medium.

The mapping of the server PFG to the client-end is done in two steps:

1. Expand the server PFG according to the bit-width of the medium. This involves the possible expansion of
both timed events and values.

2. Substitute the send/receive events with the corresponding medium WRITE/READ PFGs; PFG_{m,u} and PFG_{m,r}.

Example 3: Consider the server timed event \( t = (?a \cdot ?b) \) and a synchronous medium \( m = (?w \cdot ?z) \) (see figure 3a) with a bit-width of 8, then table 1 shows the steps involved in mapping the server PFG to the client-end for three different bit-width of \( a \) and \( b \). In the second mapping, we have to segment \( a \) and \( b \) in step 1, as they are both 16 bits wide and the medium can only transfer 8 bits at a time. In the third mapping, \( a \) and \( b \) may be combined in step 1 as they are both 4 bits wide and thus, fits into a single transfer.

The example illustrates data and event segmentation as well as data combination. In order to identify these situations, we need a way to deduce the bit-width of values, events and the medium.

Definition 6 Let \( \beta(X) \) be the number of bits necessary to represent the data contained in \( X \), where \( X \) may be an event \( (e) \), a value \( (v) \), or a medium \( (m) \):

\[
\beta(v) \triangleq \text{number of bits needed to represent value } v.
\]

\[
\beta(?a) \triangleq \beta(v)
\]

\[
\beta(?w) \triangleq \beta(v)
\]

\[
\beta(e, op e_j) \triangleq \{ \max(\beta(e_1), \beta(e_j)) \text{ for } op \in \Theta_s, \beta(e_i) + \beta(e_j) \text{ for } op \in \Theta_p \}
\]

\[
\beta(m) \triangleq \text{number of bits available in medium } m.
\]

Definition 7 The density of a transfer \( v \) in the context of a medium \( m \) is defined as:

\[
\sigma(v) \triangleq \frac{\beta(v)}{\beta(m)}
\]

i.e., \( \sigma(v) \) states how much of the medium, in terms of bits, that has to be used in order to fulfill the required transfer of value \( v \). Thus, \( \sigma(v) \) indicates whether a transfer has to be segmented \( (\sigma(v) > 1) \) or may be used in combination with other transfers \( (\sigma(v) < 1) \). These are the basic transformations involved in step 1.

As two or more values transferred in a single cycle may be viewed as a single effective value, we define the effective event as:

Definition 8 An effective event \( e_{eff} \) is an event \( e \) in which,

\[
e_{eff} \triangleq e[v_i \mid v_j/v_i, v_j]
\]

the new value \( v_i v_j \) is denoted a combined value.

From the definition of \( \sigma(e) \) we can now formulate the transformation of step 1, i.e., \( e \sim \sim e' \).

Axiom 1 (Expansion) Data segmentation:

\[
\exists e_{eff} : \sigma(v) > 1 : \; \; v \sim v_1 \triangleright v_2 \triangleright \ldots \triangleright v_n
\]

5 Hardware Generation

The original server PFG still specifies how to perform the wanted operation at the server-end of the medium. Thus, hardware is needed to store data at the server-end, and to control when enough data has been transferred in order to execute the server operation. To explain the hardware generation, consider a simple case where the communication media is a synchronous 8-bit bus able of transferring data within a single clock cycle. In this case the READ and WRITE operations take 8-bit arguments and the corresponding PFGs consists of a single timed event as shown in figure 4. The operation we want to invoke is the FMULT\(^1\) operation of figure 2b which has two 8-bit input arguments and one 16-bit output argument. The first timed event \( t_1 = (?TRF \mid ?a \mid ?b) \) of PFG_{FMULT} specifies that the two input arguments \( a \) and \( b \) should be transferred in the same cycle, along with a control code that must be assigned to the server control port in order to execute instruction \( \text{TRF} \).

However, as:

\[
t_{i,off} = (i_{TRF}) \text{ and } \sigma(i_{TRF}) = 19 > 1
\]

the media only allows one argument to be transferred in each cycle. To encompass this limitation we need a buffer and a control unit (FSM) on the server-end of the channel. The control unit will examine the output of the medium until

\[\text{where } n = [\sigma(v)].\]

Data combination:

\[
\exists v_i, op v_j : \sigma(v_i) + \sigma(v_j) \leq 1 ; v_i \triangleright v_j \parallel v_j
\]

If \( v \) is a combined value we have to consider the original values of \( v \) when doing the segmentation.

The transformation of step 2 is straight forward:

Axiom 2 (Substitution)

\[
\forall \forall v_i, v_j \in e_{eff} : \sigma(v_i) + \sigma(v_j) \leq 1 ; v_i \triangleright v_j \parallel v_j
\]

5 Hardware Generation

The original server PFG still specifies how to perform the wanted operation at the server-end of the medium. Thus, hardware is needed to store data at the server-end, and to control when enough data has been transferred in order to execute the server operation. To explain the hardware generation, consider a simple case where the communication media is a synchronous 8-bit bus able of transferring data within a single clock cycle. In this case the READ and WRITE operations take 8-bit arguments and the corresponding PFGs consists of a single timed event as shown in figure 4. The operation we want to invoke is the FMULT\(^1\) operation of figure 2b which has two 8-bit input arguments and one 16-bit output argument. The first timed event \( t_1 = (?TRF \mid ?a \mid ?b) \) of PFG_{FMULT} specifies that the two input arguments \( a \) and \( b \) should be transferred in the same cycle, along with a control code that must be assigned to the server control port in order to execute instruction \( \text{TRF} \).

However, as:

\[
t_{i,off} = (i_{TRF}) \quad \sigma(i_{TRF}) = 19 > 1
\]

the media only allows one argument to be transferred in each cycle. To encompass this limitation we need a buffer and a control unit (FSM) on the server-end of the channel. The control unit will examine the output of the medium until

\[\text{where } n = [\sigma(v)].\]

Data combination:

\[
\exists v_i, op v_j : \sigma(v_i) + \sigma(v_j) \leq 1 ; v_i \triangleright v_j \parallel v_j
\]

If \( v \) is a combined value we have to consider the original values of \( v \) when doing the segmentation.

The transformation of step 2 is straight forward:

Axiom 2 (Substitution)

\[
\forall \forall v_i, v_j \in e_{eff} : \sigma(v_i) + \sigma(v_j) \leq 1 ; v_i \triangleright v_j \parallel v_j
\]

If \( v \) is a combined value we have to consider the original values of \( v \) when doing the segmentation.

The transformation of step 2 is straight forward:

Axiom 2 (Substitution)

\[
\forall \forall v_i, v_j \in e_{eff} : \sigma(v_i) + \sigma(v_j) \leq 1 ; v_i \triangleright v_j \parallel v_j
\]
Table 1: Steps involved in mapping a server PFG to the client-end for different bit-width of data values and medium.

<table>
<thead>
<tr>
<th>bit-width</th>
<th>server PFG</th>
<th>client-end PFG</th>
</tr>
</thead>
<tbody>
<tr>
<td>m 8 8 8</td>
<td>((s_a \circ {b}))</td>
<td>((s_a \circ {b}))</td>
</tr>
<tr>
<td>8 16 16</td>
<td>((s_a \circ {b}))</td>
<td>((s_a \circ {b}))</td>
</tr>
<tr>
<td>8 4 4</td>
<td>((s_a \circ {b}))</td>
<td>((s_a \circ {b}))</td>
</tr>
</tbody>
</table>

which means that we have to expand \(t_{3,\text{off}}\) into \([\sigma(t_{\text{RES}})] = 3\) transfers. As \(t_3\) only contains 2 values, both expansion and data segmentation has to take place,

\[
\sigma(t_{\text{RES}}) = \frac{3}{8} < 1, \quad \sigma(c) = \frac{16}{8} = 2 > 1
\]

i.e., \(c\) has to be segmented into 2 value transfers, \(c_1\) and \(c_2\). Result segmentation is controlled by the server-end FSM.

A synchronization event specifies a test to be performed by the client. As the values necessary to evaluate the synchronization condition have already been acquired by means of transfers, the actual synchronization is unaffected by the choice of media. The actual implementation of a synchronization event depends on whether a part of the medium can be allocated synchronization, as assumed in [7].

Example 4: Figure 6 shows the resulting client-end PFG and implementation of the FMULT example. Formally the client-end PFG may be written as:

\[
\text{PPG}_{\text{FMULT}} = (s_{\text{TRF}} \circ \{a, b\} \circ (s_{\text{TEST}} \circ \{d, e\}))
\]

Figure 5: Channel hardware: a) overview, b) control unit FSM, c) client-end PFG.

it recognizes a control code signifying that it should buffer a number of arguments and execute a server instruction. Figure 5a shows the buffer and control unit that handle execution of the \(s_{\text{TRF}}\) instruction. Figure 5b shows the FSM that controls the buffering and server execution. Seen from the client-end, the \(s_{\text{TRF}}\) instruction has now been changed so as to execute in three cycles as shown in figure 5c. This example illustrates the main principle of the hardware generation.

The output of data from the server, as in the \(s_{\text{TEST}}\) instruction, requires that the client sends an appropriate control code to the server-end FSM. The FSM will then execute the server instruction, buffer the results and send them to the client in the following cycles. If a result is wider than the media it must be segmented at the server-end and reassembled at the client-end. For the third timed event \(t_3 = (s_{\text{RES}} || s_{\text{E}})\) we have:

\[
\sigma(t_{\text{RES}}) = \frac{19}{8} > 1
\]

6 Conclusion and Future Work

We have presented an approach to interface synthesis based on a one-sided interface model. In the context of this model, transformations involved in solving the interface synthesis problem has been presented. In particular we have focused on channel synthesis, i.e., the process of transforming the server PFG into a client-end PFG, as a direct mapping. However, interface synthesis consists of both transformations and optimizations.

If the medium is able to transfer data within a single cycle (as is the case in figure 6, one buffer and a state for each instruction may be saved as the instruction execution may take place in the same cycle as the last value transfer.

Even though the sequence of data transfers to the server is fixed, data may be send in any order over the medium, increasing the possibility of data combination, and/or giving the client synthesis the freedom to select the order. These are topics for further investigation.

Finally, our approach may be used to solve the traditional interface problem in which both client and server has been implemented prior to interface synthesis. In this case we need to introduce hardware (i.e., FSM and buffers) on the
client side in order to transform the client-end PFG back to the original server PFG, i.e., introducing an extra step in the channel synthesis.

Acknowledgements

We are grateful for comments and suggestions from Anne Haxthausen, Robin Sharp and Jørgen Staunstrup.

References


