Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation*

Bill Lin Steven Vercauteren

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium, Email:{billlin,vercaut}@imec.be

Abstract — We describe a new high-level compiler called **Integral** for designing system interface modules. The input is a high-level concurrent algorithmic specification that can model complex concurrent control flow, logical and arithmetic computations, abstract communication, and low-level behavior. For abstract communication between two communicating modules that obey different I/O protocols, the necessary *protocol conversion behaviors* are automatically synthesized using a Petri net theoretic approach. We present a synthesis trajectory that can synthesize the necessary hardware resources, control circuitry, and protocol conversion behaviors for implementing system interface modules.

1 Introduction

Today, complex digital electronic systems utilizing a wide range of IC technologies are being designed. Example technologies include application-specific ICs, FPGAs, DSPs and video processors, μ-processors, memories, system bus modules, and other off-shelf components. These different hardware and software modules can be used in combination to create complex heterogeneous hardware-software systems. These modules often interact and communicate in complex ways by transferring information and synchronizing their inputs and outputs. A key bottleneck in system integration is the design of *system interface modules*, which are special system modules required for interconnecting and synchronizing the information transfer between communicating system-level components. This is required because in general each of the communicating components may be using arbitrary and incompatible I/O protocols to communicate with its environment.

To clarify what is meant by the term "system interface module", two examples are provided to give a sense for the key issues and difficulties involved in designing interface modules. The first example, depicted in Figure 1, is an interface module that performs the data transfer from two source optical links to two destination storage buffers. The interface module first waits for a rising transition on the reset line followed by a rising transition on the start line. When this occurs, the interface module is responsible for reading 60 samples from the first optical link and inserting the samples into the first storage buffer. Concurrently, it has to read 500 samples from the second optical link and insert them into the second storage buffer. The I/O protocol used by the optical links is a simple two-phase protocol whereas the storage buffers use a four-phase protocol, as depicted in Figure 1. This example illustrates the need for concurrent control flow, asynchronous hardware resources like asynchronous counters and registers, protocol conversion from two-phase to four-phase, and low-level specification at the level of signal transitions.

The second example (taken from [16]) is shown in Figure 2. It is a system bus interface providing read and write access from the VME system bus to the shared memory port of a processor module. During a write access, the VMEbus [21] is the source of address and data information, while the processor port is the destination of address and data. In the read transfer, address information still flows from the VMEbus to the processor port, but data flows in the opposite direction. The read and write access also requires the decoding of the upper VME address bits to determine if the processor address space has been selected. Accordingly, the interface exercises

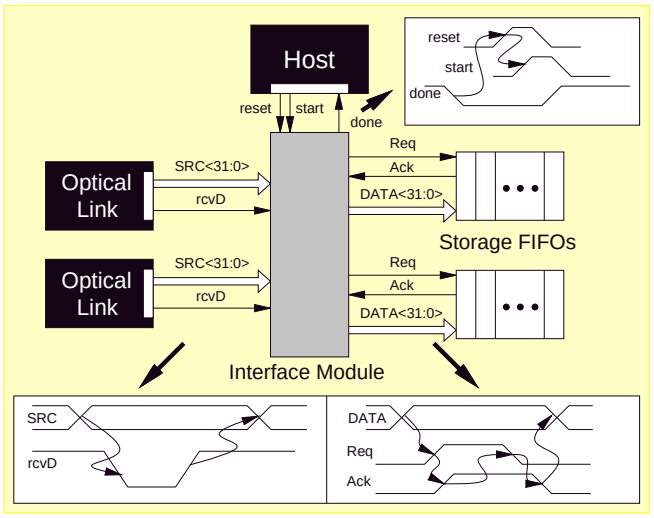

Figure 1: Interface for dual-optical source to FIFO buffers.

conditional control over the communications. To synchronize the transfer with their environment, both the VMEbus and processor port use a signaling protocol exercised on control signals, as depicted with timing diagrams in Figure 2. The protocols shown are somewhat simplified to clarify the example. The actual protocols are more complex, requiring the synchronization of address lines, data lines, and the write status signal. Both modules use different protocols, so the interface module has to also be responsible for the protocol conversion between them. This example illustrates the fact that communication actions on different "channels" may be *dependent* and *inter-related*.

In this paper, we describe a new high-level compiler called **Integral** that can be used for the automated design of such system interface modules. The input to this compiler is a high-level concurrent algorithmic specification that can conveniently be represented as an annotated Petri net model [15] extended with CSP-style rendezvous communication. With this model, complex high-level control involving concurrent threads of execution, data and information processing, abstract communication based on CSP's rendezvous model, and low-level behavior such as signal transitions (e.g. rising $s+$ and falling $s-$ transitions) may be described.

Also provided to the compiler is a library that captures the actual I/O protocols and timing information used by the different hardware-software modules in the system. These I/O protocols are specified using the signal transition graph model [6], which is also a Petri net model but with only signal transitions as possible actions. This specification of the module I/O protocol is "encapsulated" into the library. In the high-level input model, communication actions between communicating modules can be specified in an abstract way. This permits a non-expert designer to specify high-level communication between modules without a detailed understanding of the actual communication protocol characteristics employed in each individual module. For example, in Figure 2, the designer simply has to specify

*This research was sponsored in part by the European Commission under the OMI/EXACT project in ESPRIT No. 6143.

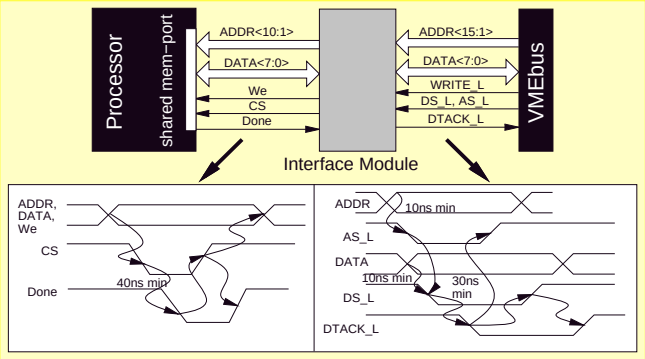

Figure 2: Interface between VMEbus and processor memory.

a "receive" operation along the VMEbus channel without specifying "how" this is achieved. The "protocol behavior" required to implement this "receive" operation will be automatically synthesized.

Given the high-level input description and a protocol library, the compiler first generates a set of *communicating* annotated Petri net models by means of direct translation. The concurrent control flow is specified by the flow-relation of the Petri net. The "transitions" of the Petri net specify abstract communication actions over *channels*, assignment operations, function calls, logical and arithmetic computations, and low-level signal transitions. Each annotated Petri net is then *successively refined* by syntactically allocating hardware resources (e.g. adders, incrementers, registers) for data related operations. This is similar to the "macromodules" approach used by existing high-level asynchronous compilers [20, 13, 5, 1], except that we only use this approach for the data processing parts. In particular, we are using "single-rail encoded" hardware modules that are organized with the data-bundling assumption [17, 5, 1]. These hardware modules can be controlled using either a *four-phase level signaling scheme* or a *two-phase transition signaling scheme*. We refer to the chosen control scheme as the *"primitive protocol"*. There is a growing consensus that the four-phase scheme is more efficient, but our synthesis trajectory permits the use of both schemes.

For internal communication actions, they are simply refined by means of handshake expansion to the internal primitive protocol (i.e. four-phase or two-phase). For *external communication actions* between communicating modules that obey different I/O protocols, the refinement step requires the *synthesis of the necessary protocol conversion behaviors*. For this synthesis task, we describe a novel Petri net theoretic approach that *implicitly* performs protocol conversion between two incompatible external I/O protocols by converting them to the internal primitive protocol. This conversion is performed automatically based on formal operations. With this approach, automatic protocol conversion between complex I/O protocols involving choices can be handled.

At the end of the refinement process, the refined Petri net contains only simple signal transitions, which corresponds to a signal transition graph model [6] that specifies the *collective control behavior*. The *control signal transition graph* is then synthesized to hazard-free gate-level logic using well-optimized low-level algorithmic synthesis methods [12, 6, 10, 11, 18, 3]. During the controller synthesis phase, *internal control signals* introduced from the refinement process may be flatten away. Thus, the synthesized solutions need not be constrained by the initial refinement structure.

An important aspect of our approach is that we use an algorithmic synthesis approach for the control processing parts instead of a syntax-directed translation approach [13, 5, 20, 1]. Circuit implementations obtained by syntax translation for control behavior tend to be highly unoptimized. This is not acceptable for the application domain considered here since a substantial portion of the behavior is control oriented. Our refinement approach effectively employs a syntax-directed approach with peephole optimizations for the data processing parts, but employs an algorithmic synthesis approach for the control parts.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 gives an overview of our specification model. The synthesis methods for the different steps in our synthesis trajectory are described in Section 4 and Section 5 : Section 4 describes the steps for synthesizing the high-level computations and the internal communication actions; Section 5 describes the steps for synthesizing the external communication actions and protocol conversion. Section 6 describes in details the application of our compilation method to design examples.

2 Related Work

Automated logic synthesis tools have been developed for the synthesis of asynchronous control circuits [3, 6, 10, 11, 12, 18]. They are based on a low-level specification at the level of binary signal transitions, e.g. signal transition graphs [6]. Though there exist powerful methods for synthesizing optimized hazard-free implementations, the specification and synthesis methods are too low-level for the problem here. Abstract communication, logical and arithmetic computations, and other high-level commands typically found in high-level programming languages, such as functional calls and hierarchy, are not supported in the specification model or by the synthesis methods. Moreover, protocol conversion behaviors are assumed to be specified by the designer, which means the designer is responsible for this difficult and error-proned design task.

High-level asynchronous compilers, mainly based on the CSP model [9], have also been developed [5, 13, 20, 1]. These compilers are based on syntax-directed translation methods. They can handle high-level logical and arithmetic computations, abstract communication, and other high-level commands. However, for the system interface design problem here, they lack several crucial capabilities: e.g., they do not perform automatic protocol conversion, which again leaves this diffcult and error-proned design task to the designer, and they generally do not handle well low-level details like signal transitions and timing, which are required for specifying low-level interface behaviors.

In [4], a compiler based on an event graph model similar to the signal transition graph model was described. The synthesis method is based on syntax-directed translation and template matching. This compiler is mainly targeted towards lower level synthesis. High-level processing and automatic protocol conversion are not handled.

In [16], a compiler designed specifically for the synthesis of (high-level) system interface modules was proposed. Their flow-graph specification model provides limited means for expressing concurrency, does not generally permit the intermixing of high-level commands with low-level details, and is not semantically well-defined. Their compilation method is based on an ad-hoc synthesis trajectory that produces a set of combinational data-path modules, a separate set of logic equations for each communication action, and an ordinary control finite state machine that is "clocked" by asynchronous signals. The main shortcoming with their compiler is that none of the hardware modules produced are ensured to be race- or hazard-free and the interactions between the modules are not ensured to be race- or hazard-free either. Their method also has trouble dealing with external protocols that have "conditional behaviors".

3 Specification Methodology

3.1 The protocol library

In order to abstract over the details of the actual I/O protocols used for communication between different hardware-software modules in the system, the I/O protocols along with relevant timing information are "encapsulated" into a "protocol library". In the high-level input model of the system interface behavior itself, the communication actions between different modules are specified in an abstract and concise way without having to specify the actual protocol details. This is because the knowledge of the protocol details are captured for the user in the underlying protocol library. In fact, the actual

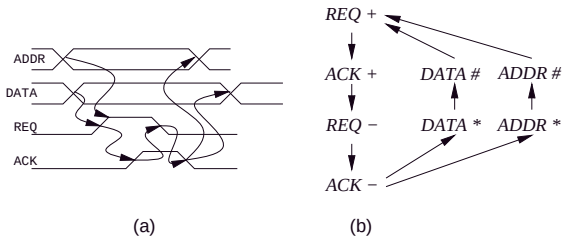

Figure 3: (a) Return-to-zero port write protocol. (b) Protocol signal transition graph for the module library.

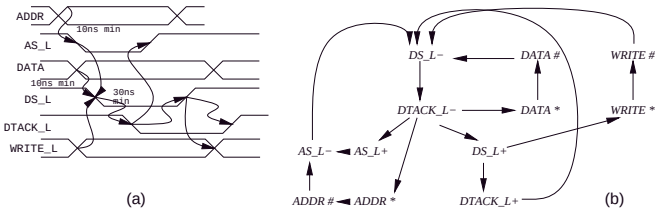

Figure 4: (a) VMEbus write protocol. (b) Protocol signal transition graph for the module library.

control signals for the protocol communication does not even appear in the high-level specification model. This way, the designer only has to specify a communication action that is required along a specified channel that uses a certain protocol. The "protocol behavior" required to implement this communication action will be automatically synthesized (cf. Section 5).

Module protocols are typically documented as informal timing diagrams in data sheets and technical manuals. For example in Figure 3, a standard "return-to-zero" write protocol is depicted. The timing diagram is shown in Figure 3(a). To formally represent this I/O protocol, we use the signal transition graph model [6], which is also a Petri net model but with only signal transitions as possible actions. This signal transition graph is referred to as the "protocol signal transition graph" of the module protocol, and it can be readily derived from the timing diagram, as shown in Figure 3(b).

Although this is a very simple protocol, it already illustrates some important (but subtle) points. This protocol is in fact used to "control" two channels : a channel for transferring the address, and a channel for transferring the data. Communication actions along these two channels must be "coordinated". In the protocol library, we explicitly associate with each protocol definition the set of "channels" that it controls. These channels are captured as arguments. In this example, the associated channels are $DATA$ and $ADDR$. The events tagged with the symbols "$*$" and "#" are used to indicate that the corresponding channel is at an "invalid" state or a "valid" state, respectively : e.g. $DATA^*$, $DATA^\#$ and $ADDR^*$, $ADDR^\#$.

In Figure 4, a more complex protocol is shown. This is only the "write" cycle of the VMEbus protocol. The protocol signal transition graph is shown in Figure 4(b). The complete protocol (will be discussed in Section 6) is considerably more complicated and involves conditional choices. In general, signal transition graphs with a general underlying Petri net model and arbitrary choices are needed to capture complex protocols. Therefore, we also use general signal transition graphs to model protocols in the protocol library. In addition, as illustrated by this example, timing information must also be annotated. They are associated with the "arcs" of the signal transition graph. For example in Figure 4, the address lines must settle at least 10ns before the falling transition of AS_L can occur. This is reflected by the associated timing annotation in the protocol signal transition graph. These timing constraints must be verified after the synthesis process, using for example the timing verification techniques reported in [2, 14, 19, 8].

```
protocol rtzRead;
protocol rtzWrite;

process shift_send :
in signal start;
channel addr⟨7:0⟩, data⟨7:0⟩: rtzRead;
channel addr_out⟨7:0⟩, data_out⟨7:0⟩: rtzWrite;
{
   boolean x⟨7:0⟩, y⟨7:0⟩;
   start+;
   addr?y || data?x;
   addr_out!y || data_out!(x≫2);
   start−;
}
```

Figure 5: A simple shift-send example.

3.2 Input specification model

We will informally introduce our high-level input specification model by means of an illustrative example shown in Figure 5. The input model is specified as a set of communicating processes over a set of CSP-style rendezvous communication channels and low-level control signals. This specification has the underlying semantics of a high-level Petri net [15] extended with rendezvous communication and the translation to an internal annotated Petri net representation is straightforward. With this model, complex high-level control involving concurrent threads of execution, data and information processing, CSP-style abstract communication, and low-level behavior such as signal transitions (e.g. rising $s+$ and falling $s-$ transitions) may be described.

Referring again to the program text in Figure 5, the command lines "protocol rtzRead" and "protocol rtzWrite" declare that the module protocols "rtzRead" and "rtzWrite" in the protocol library will be used for abstract external communication. These names identify the protocol signal transition graphs in the module library that denote these protocols. These two protocols are in fact based on the protocol depicted in Figure 3. The "process" declaration declares a module that executes concurrently with other processes in the system interface specification. A process will restart itself upon completion of its execution, implying that an operation within a process will be activated at most once during each execution of the process.

The command line "in signal" declares an input control signal named "start", and the command lines starting with "channel" declare data communication channels. The word "rtzRead" at the end of the first channel command line specifies that the fields "addr⟨7:0⟩" and "data⟨7:0⟩" are related channels that are controlled by the same module protocol "rtzRead" in the protocol library. Similarly, the word "rtzWrite" at the end of the second channel command line specifies that the fields "addr_out⟨7:0⟩" and "data_out⟨7:0⟩" are related channels controlled by the same module protocol "rtzWrite in the protocol library. Again, the detailed knowledge of these protocol are hidden in the protocol library.

In the body of this example, x and y denote internal variables, $start+$ and $start-$ specify that the process must wait for the environment to make a rising or falling transition on the wire $start$, respectively. Note that it is possible in our model to also capture low-level details at the signal transition level. This is very important in practice because the specification of system interface module, though at a "high-level", often still requires precise specification of low-level details. In our synthesis methodology (cf. Section 4), "high-level" as well as "low-level" operations are handled together.

The operators ";" and "||" mean *"Sequential composition"* and *"Parallel composition"*, respectively. $A; B$ means that first A is

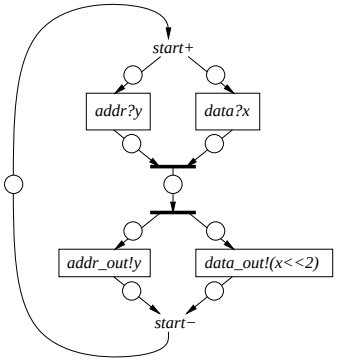

Figure 6: Annotated Petri net model for shift-send example.

executed and subsequently B. $A \| B$ means commands A and B are executed concurrently. The compound command terminates when both sub-commands are terminated. The $\|$ operator binds more than the ";" operator. In general, the commands A and B can be complex hierarchical blocks involving independent threads of control flow. Thus, complex concurrent control behavior may be specified.

The commands $addr?y$ and $data?x$ specify rendezvous "receive" operations along channels $addr$ and $data$. Unlike standard CSP, we are able to specify that the rendezvous operation must be "synchronized" with the particular protocol declared for that channel. The protocol conversion behavior is automatically synthesized by our compiler, as explained in the next section. Similarly, the commands $addr\_out!y$ and $data\_out!x$ specify rendezvous "send" operations along channels $addr\_out$ and $data\_out$.

Besides the constructs indicated already, the input model also supports standard procedural constructs like loops and if-then-else statements.

4 Synthesis I

In this section and the following section, the synthesis method of our compiler is described in detail. In the first step, the high-level input model is translated into an annotated Petri net. This intermediate form can be derived easily using a syntax-directed procedure, as can be seen in Figure 6. The parallel construction, the sequencing of actions and the choice between different actions, all have their Petri net equivalents, which makes the procedure straightforward. This annotated Petri net is then *successively refined* until all actions have been refined to simple signal transitions. In this section, we describe how the actions of the annotated Petri net are modeled, and how this is used in the refinement process to synthesize the high-level computations and the internal communication actions. Afterwards in the next section, we explain how automatic protocol conversion is performed to synthesize the external communication actions.

4.1 Internal action refinement

In Section 3, we informally introduced our specification model. This specification model is internally represented using a communicating Petri net model that we now describe more formally. Each process is a labeled Petri net $G = \langle A, P, \rightarrow, m_0 \rangle$ where A is a set of *actions*, P is the set of *places*, $\rightarrow \subseteq \mathcal{P}(P) \times A \times \mathcal{P}(P)$ is the *flow relation*, and $m_0 : P \mapsto \{0, 1\}$ is the *initial marking*.

The Petri net processes communicate and interact with each other and the environment via the communication channels, using send and receive commands, and the input/output wires, through low-level handshake signal transitions.

For each action $a \in A$, $\bullet a$ denotes the set of input places of a and $a\bullet$ denotes the set of output places. Similarly, for each place $p \in P$, $\bullet p$ denotes the set of input actions of p and $p\bullet$ denotes the set of output actions.

In our specification model, the actions in $A = E \cup D \cup H \cup R \cup Q \cup \{\epsilon\}$ can be of the following types.

$E = (I \cup O) \times \{+, -, \sim\}$ is the set of low-level signal transitions on the *input* and *output wires* I and O. s^+ and s^- denote the *rising* and *falling* transition of signal s, respectively, and $s^\sim$ denotes the *toggling* of s to its opposite value.

$D = B \times \{*, \#\}$ is the set of actions on *data wires* that specify whether the values on the corresponding set of data wires $b \in B$ are *invalid* (denoted by b^*) or *valid* (denoted by $b^\#$). b can denote a single wire or a bundle of wires corresponding to a data bus. In this case, the *width* of the data bus is abstracted.

The actions in H correspond to logical or arithmetic expressions: e.g., additions, comparisons, etc. These actions can be thought of as data processing computations.

$R = C_{int} \times \{!, ?\}(\times X)$ is the set of actions that corresponds to *internal communications*. C_{int} is the set of internal *channels* where communication takes place within a Petri net process or between two communicating Petri net processes. "!" and "?" correspond to CSP-style rendezvous *send* and *receive* operations, respectively. Data, which is optional, may be communicated over a channel using X as data variables.

$Q = C_{ext} \times \{!, ?, ?*\}(\times X)$ is the set of actions that corresponds to *external communications*. C_{ext} is the set of external *channels* where communication takes place with the environment. "!" and "?" again correspond to CSP-style rendezvous *send* and *receive* operations, respectively. Unlike internal communication actions, an *external communication protocol definition* $\pi(q)$ is associated with each external communication action $q \in Q$. Protocol conversion must be performed to communicate with the specified external protocol. We also introduce a *get* command, denoted by "?*", that is similar to the receive command except that it doesn't introduce a new communication cycle.

Finally, ϵ simply denotes a dummy transition. Function calls, hierarchy, and other high-level commands can be reduced to the above form.

Given a set of communicating Petri net processes $\Gamma = \{G_1, G_2, \ldots, G_{|\Gamma|}\}$ of the form $G = \langle A, P, \rightarrow, m_0 \rangle$, we successively refine each Petri net process until all actions have been refined to simple signal transitions, which in fact corresponds to a signal transition graph model [6] that specifies the *collective control behavior*. The *control signal transition graph* is then synthesized to hazard-free gate-level logic using well-optimized low-level algorithmic synthesis methods [6, 10, 11, 18, 3].

We now describe the action refinement process. The actions in $E \cup D \cup \{\epsilon\} \subseteq A$, corresponding to low-level signal transitions, data invalid/valid declarations, and dummy transitions, are already at their lowest level and do not need to be refined any further. However, actions corresponding to logical and arithmetic expressions, internal communication, and external communication must be further refined.

Logical and arithmetic expressions are refined using a "macro-modules" approach by means of syntax-directed translation, as in [20, 13, 5, 1]. That is, hazard-free hardware implementations necessary for performing the logical and arithmetic computations are assumed to be available in a pre-defined library. The necessary *hardware resource allocations* and their proper interconnections are done in the same way as [20, 13, 5, 1]. Compound actions are refined to more primitive actions using an action grammar. The interested reader can refer to these works for details. In particular, we are using "single-rail encoded" hardware modules that are organized with the data-bundling assumption [17, 5, 1]. These hardware modules can be controlled using either a *four-phase level signaling scheme* or a *two-phase transition signaling scheme* (cf. Figure 7). We refer to the chosen control scheme as the "*primitive protocol*". These protocols are used to trigger the computations and to detect their completions. For the sake of exposition, we will use the four-phase level signaling scheme as the primitive protocol for the remainder of the paper, but either scheme can be used. Using the single-rail encoded modules with the data-bundling assumptions, delays may need to be inserted to ensure proper handshaking. This is a well-studied problem [17, 5, 1] and we are using similar techniques. However, unlike [20, 5, 1], the control logic is not syntactically allocated, but the corresponding

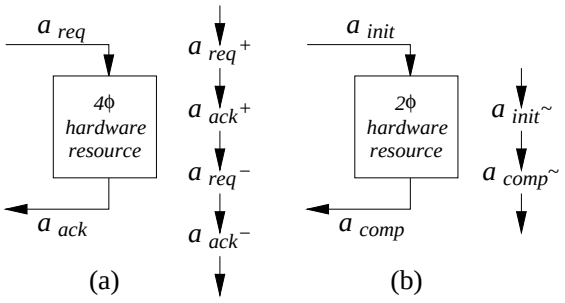

Figure 7: Hardware resource blocks. (a) 4-phase. (b) 2-phase.

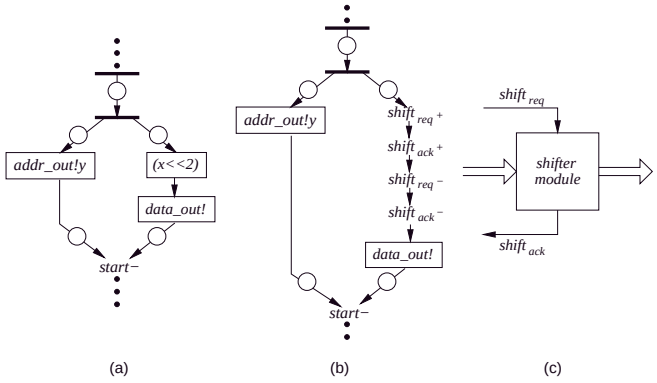

Figure 8: Example expression refinement for the shift-send example.

control behavior is refined into the Petri net itself, which is then later synthesized algorithmically at the end of the refinement process.

In Figure 8, an expression refinement is shown. An asynchronous shifter is allocated. The corresponding control signal transitions are shown in the partially refined annotated Petri net of Figure 8.

The refinement of internal communication actions is done in a similar way as expression refinement. In particular, "send" and "receive" commands are refined to the corresponding primitive protocol sequences. The expanded primitive protocol sequence, e.g. the four phase protocol, actually implements a full rendezvous handshake protocol. The necessary registers and multiplexing hardware are allocated for implementing the transfers.

5 Synthesis II : Protocol Conversion

After the refinement of expressions and internal communication actions (cf. Section 4), we are left with a refined Petri net that only has low-level actions and external communication actions. In this section, the refinement of external communication actions and the problem of protocol conversion are discussed. We first informally introduce our protocol conversion method by means of examples. Then, we will describe the procedures and operations more formally.

5.1 Examples

Consider first the example depicted in Figure 9. We wish to refine the external communication action $D!x$, which corresponds to the "sending" of x along the channel D from the interface module to the external module. For this action, the interface module is in fact the "sender" and the external module is the "receiver". The external protocol π (cf. Section 3.1) used by the external module to control channel D is shown in Figure 9(b). In this external protocol definition, the pair of events "D^*" and "$D^{\#}$" are used to indicate when the data on the channel is "invalid" and "valid", respectively. When the data is valid for the external module to safely "receive", it expects the "sender" to *raise* its input wire $STRB$. When it has processed the data, it *toggles* its output wire $SACK$ to acknowledge to the sender that it has read the data, and the protocol cycle is completed when the sender *lowers* $STRB$. Using this protocol, the external module assumes the data is "invalid" in the region from D^* to $D^{\#}$, but it assumes that the data is "valid" in the region from $D^{\#}$ to D^*, and hence can be safely processed.

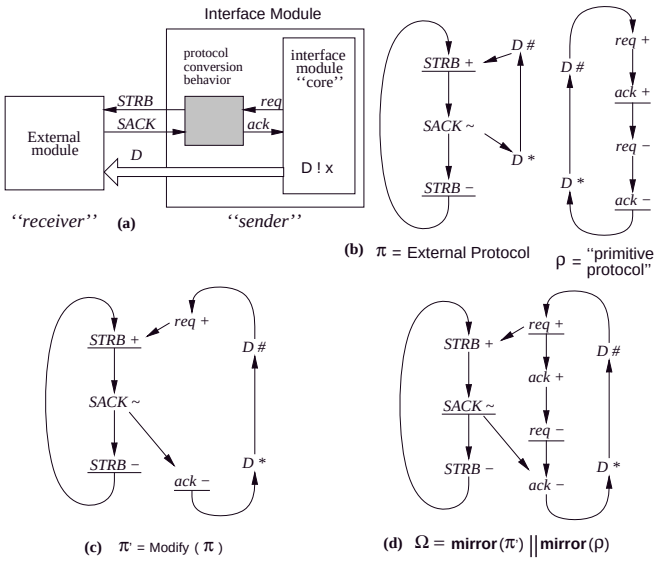

Figure 9: Simple conversion example. Part (d) is the *synthesized Protocol conversion Signal Transition Graph* Ω.

The refinement of an external communication action is achieved in two steps. First it is refined in the specification graph G in the same way as an internal communication action, i.e., by means of handshake expansion into a "primitive protocol" sequence; in this case, we are using the 4-phase handshake scheme as the primitive protocol with the wires req and ack. The primitive protocol signal transition graph ρ for a "send" operation is shown in Figure 9(b). It also has a pair of events with the names "D^*" and "$D^{\#}$" to indicate when the data on the channel is "invalid" and "valid", respectively. In the second step, we synthesize a *protocol conversion signal transition graph* Ω that specifies the protocol conversion behavior for converting between the external protocol π and the internal primitive protocol ρ.

The basic idea in our protocol conversion synthesis method is "compose" the "mirror" of the external protocol π with the "mirror" of the internal primitive protocol ρ, synchronizing on the data events D^* and $D^{\#}$. The "mirror" of the external (internal primitive) protocol π (ρ) simply results in an interchange of the input and output wire sets. This is because the "output" of the external (internal primitive) protocol is in fact the "input" of the protocol conversion specification, and vice versa, as shown in Figure 9(a). Parallel composition [7] implicitly computes the protocol conversion behavior. Both of these operators are defined in Appendix A. This is nearly correct. However, there is a problem in the interpretation of π. Following this protocol graph, the protocol conversion "behavior" has to raise the strobe line $STRB$ as soon as it knows the data on the channel is really "stable", hence "valid". But the protocol conversion "behavior" cannot know when exactly the channel has become stable until it has received an input req^+ from the interface module "core" confirming this event. In the same way, the interface module core may not lower the ack line before the external module is ready for it, i.e. after lowering the ack line the sender can invalidate the channel D, while the receiver may still be processing it.

In Section 5.4, we describe a systematic procedure for modifying the external protocol so that the composed behavior corresponds

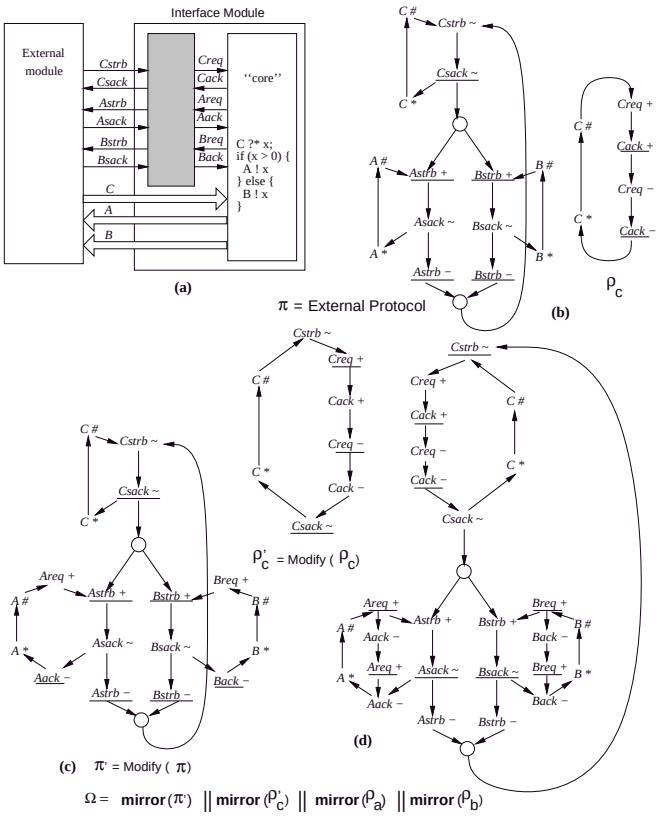

Figure 10: Multi-channel example with choice. Part (d) is the *synthesized protocol conversion Signal Transition Graphs* Ω.

exactly to a correct protocol conversion behavior. In general, it is the "receiver's" protocol that must be modified. The modified external protocol π is shown in Figure 9(c). If we compose the modified "mirrored" behavior of the modified external protocol with the "mirrored" behavior of the primitive protocol, we get the desired protocol conversion signal transition graph, as shown in Figure 9(d). Once we have synthesized the desired protocol conversion signal transition graph, we can compose it with the refined specification graph of the interface module so that the protocol conversion behavior is incorporated.

Consider now the second example shown in Figure 10. We have to refine three communication actions that are in fact related because they are being controlled by the same protocol. The external protocol π, shown in Figure 10(b), involves "choices". It acts as a "sender" along channel C, but it acts as a "receiver" along channels A and B. To read channel C, we use the *get* command ($C?^*X$) instead of a receive command to indicate that, when sending the register X back to the external module, we want to "talk" to the same external protocol communication cycle. In fact, if specify e.g. $C?^*X; C?Y$ then the contents of register X and Y will always be the same. If however we specify e.g. $C?X; C?Y$, the contents of X and Y in general may not be the same, because after the first receive command, a new communication cycle will start, possibly placing some different data on channel C.

The external module is the "receiver" along channel A and B, so we can apply the same modifications to the protocol signal transition graph as mentioned in the example of Figure 9. This can be seen in Figure 10(c). For channel C, the interface module acts as the "receiver". In this case, the primitive protocol has to be modified in the same way, as can be seen also in Figure 10(c). The modified primitive protocol is denoted as ρ'. After mirroring and composing, the resulting synthesized protocol conversion signal sransition graph then looks like Figure 10(d).

5.2 Overall Procedure

In this section, the protocol conversion method is described more formally. In the first step, the external communication actions are refined in the same way as the internal communication actions. Afterwards all external communication actions are grouped into sets of actions that are "interacting" on a particular protocol instance. These sets are called *clusters* and together with the appropriate protocol instances, they build up the protocol conversion "behaviors" (cf. examples). For each protocol conversion "behavior", the "receivers" are adapted to the "sender", by introducing some event anchors. Afterwards, the behaviors of the "receivers" can be composed with the behavior of the "sender", to become the desired protocol conversion behaviors. In order to become the behavior of the complete interface module, the behavior of the interface module "core", derived by refining all actions to low-level actions, must be composed with all the necessary protocol conversion behaviors. This results into the following algorithm.

Algorithm 5.1 *(Protocol Conversion)*

RefineExtComm (G) {
  Σ = *FindClusters* (G);
  For each $\sigma \in \Sigma$ {
    let π *be the external protocol to control* σ;
    let $\Psi = \emptyset$;
    For each $q \in \sigma$ {
      G = *Refine* (G);
      create primitive protocol ρ;
      (π, ρ) = *SetEventAnchors* (π, ρ);
      $\Psi = \Psi \parallel \mathbf{mirror}(\rho)$;
    }
    $\Omega = \Psi \parallel \mathbf{mirror}(\pi)$;
    $G = G \parallel \Omega$;
  }
  Return (G);
}

In the following sections, the functions used in Algorithm 5.1 will be outlined.

5.3 Find clusters

In the first step of the procedure, we have to identify all communication actions that are "interacting" on a particular external protocol instance. The set of all these communication actions is called a *cluster*. We say that all elements of a cluster belong to the same *protocol communication cycle*. In order to find these clusters, we first have to label each external communication action q with a instance number $L(q)$ of the external I/O protocol it is communicating with. The labeling algorithm reduces to a breadth-first traversal of the Petri net specification graph, keeping a counter that runs over the instance numbers of each external I/O protocol. A cluster for a certain protocol j with instance number i can then be defined as follows:

$$\sigma_j^i = \{q_n \in Q \mid \forall q_1, q_2 \in \Sigma_j, L(q_1) = L(q_2) = i \text{ and } \pi(q_1) = \pi(q_2) = j\}$$

We define Σ simply as the set of all these clusters.

5.4 Make synchronization

In this section some synchronization points are introduced. In order to get a correct data transfer, the "receiver" must know exactly when the data on the channel is valid and hence can be processed. The "receiver" must also make sure that it has completely finished reading the channel, before it allows the "sender" to disable it. Therefore, we have to modify the the "receiver" part of the protocol to adapted it to its "sender" counterpart.

Algorithm 5.2 *(Make Synchronization Points)*

SetEventAnchors (π, ρ) {
  let D *the communication channel for* ρ;
  If ρ *is sender* {
    (π, ρ) = *ModifyProtocol* (π, ρ);
  } *Else* {
    (π, ρ) = *ModifyPrimitive* (π, ρ);
  }
  Return (π, ρ);
}

Both the external module and the primitive protocol of the interface module "core" can act as a "receiver". These two possibilities are treated in the following sections.

5.4.1 Modifying the external protocol

In this case the external module is the "receiver" and the primitive protocol is the sender. In the external protocol signal transition graph, different actions are specified to happen after the channel has become stable. However, the "receiver" cannot know when the channel has become stable until it has received a $\rho_{req}+$ from the primitive protocol. So, any action that is specified to occur after $D^{\#}$ in the protocol signal transition graph can only occur after $\rho_{req}+$. Therefore we have to replace every instance of a $D^{\#}$ event by the same $D^{\#}$ event followed by a $\rho_{req}+$ event. Another problem is that the "receiver" must have finished reading the channel before the $\rho_{ack}-$ event of the sending primitive protocol occurs. Therefore we have to replace every instance of a D^{*} event by a $\rho_{ack}-$ event followed by the same D^{*} event. The algorithm can be defined as follows.

Algorithm 5.3 *(Modifying the external protocol)*

ModifyProtocol ($\pi = \langle A_\pi, P_\pi, \rightarrow_\pi, m_{0_\pi}\rangle, \rho = \langle A_\rho P_\rho, \rightarrow_\rho, m_{0_\rho}\rangle$) {
  For each $D^{\#}, D^{*} \in \pi$ {
    $\rightarrow_\pi = \rightarrow_\pi - (\bullet D^{\#}, D^{\#}, D^{\#}\bullet) - (\bullet D^{*}, D^{*}, D^{*}\bullet)$
      $\cup\ (\bullet D^{\#}, D^{\#}, p_{actv})$
      $\cup\ (p_{actv}, \rho_{req}+, D^{\#}\bullet)$
      $\cup\ (p_{term}, D^{*}, D^{*}\bullet)$
      $\cup\ (\bullet D^{*}, \rho_{ack}-, p_{term})$;
    $A_\pi = A_\pi \cup \rho req^{+} \cup \rho ack^{-}$;
    $P_\pi = P_\pi \cup p_{actv} \cup p_{term}$;
  }
  Return (π, ρ);
}

5.4.2 Modifying the primitive protocol

In this case the external module is the "sender" and the primitive protocol is the receiver. In fact, this is nearly the same algorithm as in Section 5.4.1, but with the arguments interchanged. The only difference is that in Section 5.4.1, the primitive protocol is acting as the sender, and thus only one $\rho_{req}+$ event and one $\rho_{ack}-$ are possible. Because in this case the external module is the "sender", possibly having a very complex protocol structure, different events can confirm the becoming stable of a channel, and different events can lead to disabling of the channel. Therefore we have to pick only one possible combination of input events confirming the channel becoming stable, and one possible combination of output events leading to the disabling of the channel. With these assumptions, the algorithm is in fact the same as Algorithm 5.3 and can be defined as follows.

Assumption 5.1 *Let* D *the communication channel for* ρ*. Then for all fanout places of all* $D^{\#}$*, all successor actions must be output transitions.*

Assumption 5.2 *Let* D *the communication channel for* ρ*. Then for all* D^{*}*, there must exist at least one fanin place such that all its precessor actions are input transitions.*

Algorithm 5.4 *(Modifying the primitive protocol)*

ModifyPrimitive ($\pi = \langle A_\pi, P_\pi, \rightarrow_\pi, m_{0_\pi}\rangle, \rho = \langle A_\rho P_\rho, \rightarrow_\rho, m_{0_\rho}\rangle$) {
  $\rightarrow_\rho = \rightarrow_\rho - (\bullet D^{\#}, D^{\#}, D^{\#}\bullet) - (\bullet D^{*}, D^{*}, D^{*}\bullet)$
    $\cup\ (\bullet D^{\#}, D^{\#}, p_{actv})$
    $\cup\ (p_{term}, D^{*}, D^{*}\bullet)$;
  For each $D_i^{\#} \in \pi$ {
    Select a fanout place $p \in D_i^{\#}$;
    For each $t \in p\bullet$ {
      $\rightarrow_\rho = \rightarrow_\rho \cup (p_{actv}, t, \bullet\rho_{req}+)$;
      $A_\rho = A_\rho \cup \{t\}$;
    }
  }
  For each $D_i^{*} \in \pi$ {
    Select a fanin place $p \in D_i^{*}$;
    For each $t \in \bullet p$ {
      $\rightarrow_\rho = \rightarrow_\rho \cup (\rho_{ack}-\bullet, t, p_{term})$;
      $A_\rho = A_\rho \cup \{t\}$;
    }
  }
  $P_\rho = P_\rho \cup p_{actv} \cup p_{term}$;

  Return (π, ρ);
}

```
extern function decode (address); /* addr dec */
protocol vme; /* VME protocol */
protocol mem; /* memory protocol */

process vme_to_mem :
channel addr1⟨14:1⟩, data1⟨7:0⟩, write : vme;
channel addr2⟨10:1⟩, data2⟨7:0⟩, we : mem;
{
   boolean mode, x⟨14:1⟩, y⟨7:0⟩, z⟨7:0⟩;
   write?*mode || addr1?*x ;
   if (decode (x⟨14:11⟩) == 1) {
      if (mode == 0) { /* read mode */
         addr2!x || we!mode || { data1?y ; data2!y }
      } else { /* write mode */
         addr2!x || we!mode || { data2?z ; data1!z }
} } }
```

Figure 11: Specification of VMEbus to processor memory interface.

6 Implementation and Experience

The techniques described in this paper have been implemented in a new high-level compiler called **Integral**. This compiler is linked to our asynchronous circuit compiler called **Assassin** [12]. **Assassin** can synthesize from a signal transition graph description to a hazard-free gate-level design. We use this compiler to synthesize the control and protocol conversion behaviors derived from the **Integral** compiler.

We have worked out a number of actual-design examples. In this section, we show the VME bus interface example described in Section 1. The purpose of this experiment is to show that our method can handle the communication between multiple modules having arbitrary protocols, and indeed provides a powerful mechanism for connecting and synchronizing modules into a system.

The program text is shown in Figure 11. The I/O protocols of the full VME system bus and the memory port are shown in Figure 12. Note that the full VME protocol involves *choices*.

After the refinement process, the resulting control signal transition graphs are then synthesized to the gate-level using our asynchronous

circuit compiler **Assassin** [12, 18, 10]. The resulting circuit implementation is shown in Figure 12.

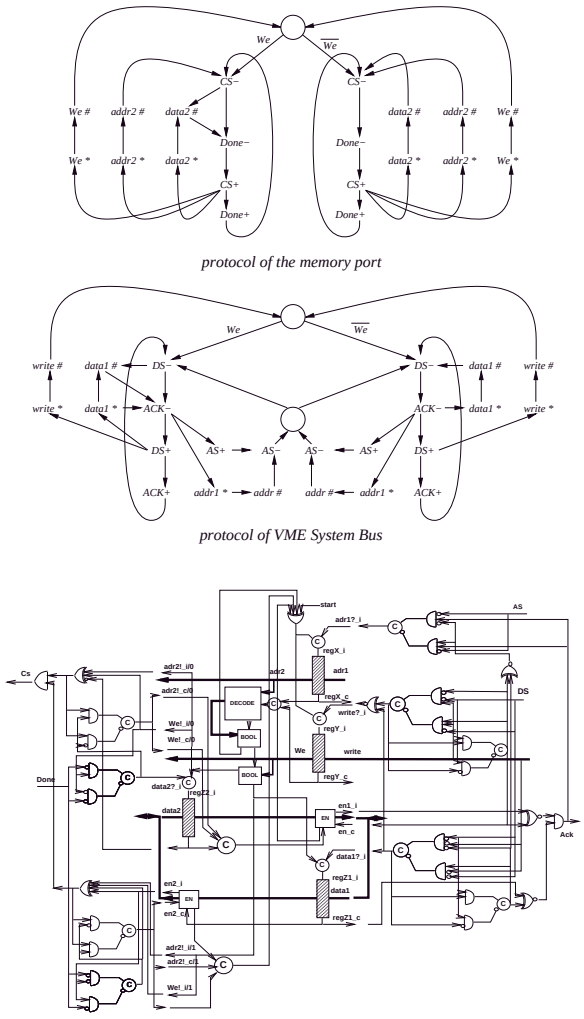

Figure 12: I/O protocol of the full VME system-bus and the memory protocol. Synthesized implementation

A Petri Net Algebra

The parallel composition of two general Petri nets is defined below. It is defined to be the synchronization of common actions. This composition works on general nets with possibly "multiple occurrences" of the same action. The operator works without explicit unfolding, but produces the necessary replication of actions and places implicitly for correct construction.

Definition A.1 (Parallel Composition) [7] *Let for $i \in \{1,2\}$, $G_i = \langle A_i, P_i, \rightarrow_i, m_{0_i} \rangle$ be two Petri nets with $P_1 \cap P_2 = \emptyset$. The* **parallel composition** *of two nets is defined as:*

$$G_1 \| G_2 = \langle A_1 \cup A_2, P_1 \cup P_2, \rightarrow, m_{0_1} \cup m_{0_2} \rangle$$

where

$$\begin{aligned} \rightarrow \quad = \quad & \{(PI, a, PO) \in \rightarrow_1 \cup \rightarrow_2 \mid a \notin A_1 \cap A_2\} \cup \\ & \{(PI_1 \cup PI_2, a, PO_1 \cup PO_2) \mid a \in A_1 \cap A_2 \wedge \\ & \qquad (PI_1, a, PO_1) \in \rightarrow_1 \wedge (PI_2, a, PO_2) \in \rightarrow_2\} \end{aligned}$$

When some of the actions are "events" of inputs and output wires, then the parallel composition of two nets is only defined when they have disjoint output wires. Let I_1 and O_1 be the inputs and outputs wires, respectively, for G_1, and I_2 and O_2 be the inputs and outputs wires, respectively, for G_2. Then in the composed Petri net $G = G_1 \| G_2$, $O = O_1 \cup O_2$ and $I = (I_1 \cup I_2) - O$.

Definition A.2 (Mirror) *Let $G = \langle A, P, \rightarrow, m_0 \rangle$ be a Petri net. Let (I, O) be a set of input and outputs wires associated with G. The* **mirror** *of G, denoted as* **mirror**(G)*, is simply the same Petri net, but with (O, I) as the input and output wires, respectively.*

The mirror operator simply interchanges the input and output wire sets. A full Petri net algebra has been developed and described in [7]. The interested reader is referred to that paper for details.

References

[1] V. Akella and G. Gopalakrishnan. Shilpa: a high-level synthesis system for self-timed circuits. In *International Conference on Computer-Aided Design*, November 1992.

[2] T. Amon, H. Hulgaard, S. Burns, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. In *IEEE International Conference on Computer Design*, October 1993.

[3] P.A. Beerel and T. Meng. Automatic gate-level synthesis of speed-independent circuits. In *International Conference on Computer-Aided Design*, November 1992.

[4] G. Borriello. *A new interface specification methodology and its application to transducer synthesis.* Ph.D thesis, University of California, Berkeley, May, 1988.

[5] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *International Conference on Computer-Aided Design*, November 1989.

[6] T.-A. Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. Technical Report MIT-LCS-TR-393, 1987.

[7] G. de Jong and B. Lin. A communicating Petri net model for the design of concurrent asynchronous modules. In *ACM/IEEE Design Automation Conference*, June 1994.

[8] D. Doukas and A. S. LaPaugh. Clover : A timing constraints verification system. In *28th ACM/IEEE Design Automation Conference*, June 1991.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, pages 666–677, August 1978.

[10] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *ACM/IEEE Design Automation Conference*, June 1994.

[11] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *ACM/IEEE Design Automation Conference*, June 1991.

[12] Ch. Ykman Couvreur, P. Vanbekbergen, and B. Lin. *Assassin: An Asynchronous I/O Interface Synthesis System.* Tutorial and reference manual. IMEC Lab., October 1993.

[13] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1:226–234, 1986.

[14] K. McMillan and D.L. Dill. Algorithms for interface timing verification. In *IEEE International Conference on Computer Design*, October 1992.

[15] J.L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.

[16] J. S. Sun and R. W. Brodersen. Design of system-level interfaces. In *Proceedings of the International Conference on Computer-Aided Design*, November, 1992.

[17] I. E. Sutherland. Micropipelines. *Communications of the ACM. The 1988 ACM Turing Award Lecture.*, June 1989.

[18] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A generalized state assignment theory for transformations on signal transition graphs. In *International Conference on Computer-Aided Design*, November 1992.

[19] P. Vanbekbergen, G. Goossens, and H. De Man. Specification and analysis of timing constraints in signal transition graphs. In *European Design Automation Conference*, March 1992.

[20] K. van Berkel. *Handshake circuits: an intermediary between communicating processes and VLSI.* Ph.D thesis, Philips Research Laboratories, Eindhoven, The Netherlands, 1992.

[21] VITA. *VMEbus specification manual.* PRINTEX Publishing, 1985.