Institute for Software Integrated Systems
Vanderbilt University
Nashville Tennessee 37235

VANDERBILT UNIVERSITY
INSTITUTE FOR SOFTWARE INTEGRATED SYSTEMS

TECHNICAL REPORT

TR #:	**ISIS-01-200**
Title:	**Interface Refinement and Synthesis for Component-Based Design**
Author:	**James Tuck, Ted Bapty**

Abstract

In component-based design, the functionality of a system is distributed among independently defined components. Interaction between components occurs through well-defined interfaces that link components together. However, the same abstraction that yields such strength can also hinder design when components cannot support an interface or are inefficiently wrapped to achieve compatibility. For the full benefits of component-based design to be realized, the performance penalty of incompatible interfaces must be removed.

A model sufficient to describe diverse interfaces and a tool capable of integrating these interfaces are sought to remove the performance penalty commonly found in component based systems. The work and experience from the Adaptive Computing Systems project serves as the foundation for this approach.

KEYWORDS
Component-Based Design, Interface Synthesis, HW/SW Synthesis, FPGA, VHDL, Design Environment, Model-Integrated Computing.

ACKNOWLEDGMENTS
This work was sponsored by the Defense Advanced Research Projects Agency, Information Technology Office, under contract #F30603-96-2-0227.

TABLE OF CONTENTS

INTRODUCTION

In component-based design, the functionality of a system is distributed among independently defined components. Interaction between components occurs through well-defined interfaces that link components together. As a result, communication issues are effectively separated from computational issues. The separation of data transfer from computation minimizes inter-component interactions, which facilitates synthesis of large-scale, complex systems. In addition, system verification/validation can be separated into V&V of component and V&V of the composed system. The ACS Model Integrated Development Environment (ACSMIDE) [1] implements a component-based design methodology to design, simulate, and synthesize complex, dynamically reconfigurable hardware/software systems.

In such environments, the form and structure of interfaces can have a significant impact on how components and systems must be designed. To simplify system construction, a small set of 'standard' interfaces is typically adopted. This greatly simplifies the automatic construction of systems, since the system generator only needs to match interfaces from a small set of possibilities. Also, components must conform to one of the interfaces in the set. Often, however, these interfaces limit how the approach can be applied, or can impose unacceptable performance penalties.

Component-based design solves many problems in heterogeneous architecture design and hierarchical design, and becomes more valuable as system complexity increases. However, for these benefits to be realized, the performance penalty must be removed. Specifically, an approach is sought that allows components to maintain their native interfaces because they are most efficient for the specific component. Since this will result in a multitude of different interfaces residing within a system, the system integration tools must manage the interconnections between incompatible protocols. The end result would allow rapid, automatic integration of systems while retaining the efficiency of components.

INTERFACES IN ACSMIDE

The ACSMIDE implements a hierarchical decomposition for system design. The functional *primitive* implements the 'leaf nodes' of the hierarchy and is the basic functional unit. A primitive maps down to a single function or entity on a given resource. It may represent a look-up-table in memory, a thread on a processor, or a multiplier on an FPGA. In a design, a primitive on an FPGA may be connected to a primitive on the same FPGA, on another FPGA, or on a software-programmed processor. Connections that span across resources, as an FPGA primitive to a processor primitive, are supported via layered protocols.

There are two distinct layers: the component level interfaces and resource level interfaces. The higher layer describes the type of protocols that exist between two components in the structural dataflow. The lower layer describes the protocols between hardware resources. These two layers meet when the dataflow must pass across resource boundaries. As a result, there are interfaces that define how a component level interface connects to the lower level hardware. On any given resource, there are interfaces that connect components, and interfaces that perform low level I/O. To support hybrid connections, the following analysis is performed:

1. Do the two connected components use compatible protocols?
2. Consider each component's interface. For each component, is there a compatible interface on the parent hardware to perform the necessary I/O operations?
3. For each resource being connected, are there compatible interfaces that allow inter-resource communication?

By insuring interface connectivity at each layer, from the component down to the hardware layer, a hybrid connection may be verified and generated. The only problem with this method is that very few components, especially those from different vendors, have similar interfaces. The requirement that all components must share the same interface can restrict the set of available components, sometimes to a single vendor.

In the ACSMIDE, for example, the current solution wraps the components with additional hardware to emulate a specific common interface. (N.B. It is possible to add new interface types to the ACSMIDE. However, doing so requires expert knowledge of the system, and an end user cannot be expected to have that knowledge. Instead, components are wrapped with previously available interfaces.) This layer makes each component compatible with any other wrapped component. The current interfaces support an 8, 16, or 32-bit parallel connection, with an asynchronous handshake protocol. While this results in complete component interoperability, a price is paid in both speed and space efficiency. In some cases, particularly large-grained computation, the overhead is negligible, but when the communication timing is on par with computation time, (i.e. small grained) the extra cost is unacceptable.

Furthermore, hardware designs achieve efficiency by integrating communications with computations; consequently, computation and communication can actively work at the same time. However, any time a system is broken up into components, the extent to which communications and computations can interlock is diminished. In an attempt to avoid further loss of efficiency due to wrapping components, the ability to integrate components that use different interfaces is desired. Rather than shoehorning every component into a particular shape and risk further inefficiency, or maintaining a library of interfaces which can never be complete, allow each component to keep its own native interface and automatically generate active connectors between incompatible interfaces. This will allow integration of unmodified, off-the-shelf components, and avoids the potential inefficiency of 'wrapped' interfaces.

In order to implement this capability in a component-based design environment, it is necessary to understand the parts of a protocol. Consider the connection between components '**FROM**' and '**TO**'. The signal represents a data flow from an output port in **FROM** to an input port in **TO**. The **Data** is the information payload of the communication. This simple connection may actually represent a write operation to memory. As a result, the integration tools must know that along with the data being written to memory, there must be signals for the address and control lines. Because so much information is hidden in a dataflow diagram of a system, information specific to each interface must be captured and made available to the integration tools. Hence, a model is sought to represent the interfaces used in the system. In the ACSMIDE, interfaces are considered to be centered around a data signal. As in the memory example above, the address, read, write, and chip enable signals were generated for the single data signal. Keeping with the ACSMIDE assumption, a data centered view of interfaces will be developed in this paper.

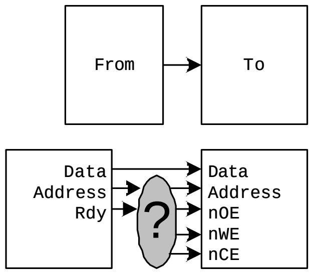


Figure 1: Dataflow vs. Implementation

Commonly used components were considered in an attempt to define the basic parts of an interface. Most components developed for ACS utilized several handshake signals to pass data from one component to another. However, many third-party components use a single signal to indicate data is ready; such components must be continually watched to insure no data is lost. Because these components are precisely what the interface model is targeting, it is imperative that their operation be supported as well. Some very simple rules came out of studying these components.

The first requirement is that the ports have the same data type on both sides of the connection (i.e. we are not attempting to do data conversion). Also, there must be some definition as to the signals involved in the handshaking typically found in hardware protocols. In a dataflow model, only data signals are drawn, but other signals will participate in the protocol. So, a simple list of requirements for a protocol is as follows:

- A required *primary data signal* on which the data is transferred and around which the protocol is centered.
- Optional non-control signals, which must be preserved across protocol boundaries (i.e. the address signal shown above) carrying data dependent on a primary signal.
- Optional control signals which may be input or output (i.e. nOE, nWE, nCE).
- Defined beginning and end.

LITERATURE REVIEW

Various methods for interface representation and protocol synthesis have been presented in the literature.

Narayan and Gajski define interface process generation as generating "an interface between two communicating processes with fixed but incompatible protocols." [2] An interface process would insure the proper transfer of data by performing the necessary actions on either side in an order to guarantee proper transfer of data. Narayan points out that the descriptions of the protocols should be enough for the generation of the complete interface process. He noted that in previous work, tools required signals on either component to have the same name or that merge tags be inserted on corresponding event graphs so that signals be appropriately matched. Ideally, the user should not be forced to guide the integration tools for synthesis to be successful. As a result, Narayan focuses on inferring protocols from the HDL used to design the component, and he defines types of statements and how to interpret them for proper analysis. Performing protocol analysis on an HDL is worthwhile if the source code is available; however, many components are sold only as a netlist. Also, it is entirely possible that someone may design the protocol in a way not anticipated in his analysis scheme. For these reasons, it seems sensible to look for a protocol representation separate from the actual implementation of the component.

Seeking alternative methods for representing protocols, Johnny Öberg [3] showed how a grammar based specification language can be used to synthesize data communication protocols. Öberg points out that languages like VHDL and C++ "have evolved to support computation intensive applications… Communication in such languages are weakly supported using data structuring elements to specify the interface and computation elements to specify the implementation of protocol for communication between systems." In particular, Öberg shows how a specified grammar may generate a hardware implementation as efficient as a hand coded design developed by an experienced designer. Öberg's work does not focus particularly on incompatible protocol synthesis, but it does support seeking representations for protocols that act as formal descriptions of the designed components.

Passerone et al [4] used regular expressions to represent protocols. By converting the regular expressions to pseudo-deterministic finite automata, product state analysis may be used to perform synthesis of interfaces between incompatible protocols. To distill the essence of Passerone's method, consider the components shown in the figure. Protocol 'a' is communicating with protocol 'b'. A potential solution to connect the components could work as follows. The first component passes its data to the intermediate component using protocol 'a'. Protocol 'a' finishes its transaction then initiates protocol 'b', which completes the process. However, this result is sub-optimal because the latency is the sum of that for 'a' and 'b'. Instead of sequential execution, it is possible for 'a' and 'b' to overlap. Using the criteria "do I have enough data to proceed" allows optimization of the 'a' and 'b' product space. Passerone states the parameters as follows: never output a piece of data that has not yet been received, transfer all the data, and minimize the latency. Using these rules, an implementation that looks as shown is achieved.

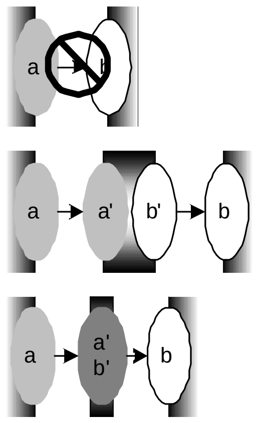


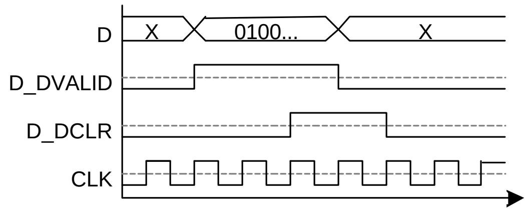
Figure 2: Steps for interface generation

INTERFACE MODEL

Drawing on previous work, an interface description should capture pertinent information regarding the signals in a protocol, the type of data being handled, and how the signals interact to pass the data. Therefore, an interface is a 3-tuple**, (D,K,P)** where **P** is a regular expression which defines the protocol, **D** is the set of signals, and **K** is a subset of **D** indicating those signals whose data must be preserved across the interface.

The set, **D**, will contain every signal that is involved in the transfer of data. For example, a TI comm-port has five signals that would be included in this definition: NCSTB, NCRDY, NCACK, NCREQ, and DATA. NCSTB and NCRDY perform data handshaking; NCACK and NCREQ control port direction; and DATA transmits the information payload. Each of these signals must be present because the value of each signal has a bearing on the state of the interface.

P, a regular expression, describes the sequence of values permissible over **D** at any given point in time. Passerone [4] describes a language for programming protocols. In his language, an n-tuple of signals with a specific type is defined. Afterwards, a regular expression is written with the different possible values of the n-tuple as the alphabet. Consequently, a protocol is the language defined by the set of successful sequences of data defined for the signals in **D**. So, what is captured by this grammar is much like a waveform diagram. Each token in the regular expression corresponds to a place on the waveform diagram. Consider the waveform for the DVALID/DCLR protocol defined below. The waveform diagram to the left represents a protocol commonly used between hardware components on an FPGA in applications built on ACSMIDE. This protocol is typically used to transfer data in parallel in one cycle. D_DVALID is an output; when it goes high, the other component knows data is available. D_DCLR is an input indicating the data has been latched. The state machine for the protocol, as implemented by the interface description above, is shown as well. The signals participating are D,D_DVALID, and D_DCLR. D carries the information payload and must be preserved across protocol boundaries. The grammar that describes the waveform is displayed as well. Note that the specified grammar accurately portrays all possible sets of values that data on the signals may assume.



Format: {D, D_DVALID, D_DCLR}

Regular Expression:
{X, 0, 0}*
{D,1,0}+
{D,1,1}
{X,0,1}

Figure 3: DCLR/DVALID Protocol Representation

The regular expression written for **P** cannot explain why or when a transfer will occur, but it completely specifies how the transfer will take place on the signals. It is very common for engineers to communicate the functionality of protocols and interfaces by the use of waveforms. Capturing this description in a regular expression makes it easy to use the documentation provided with any hardware component to write the appropriate description. Also, because a regular expression defines the transaction from neither the viewpoint of sender or receiver, the same expression can be used for either a master or slave.

INCOMPATIBLE INTERFACE SYNTHESIS

Incompatible interface synthesis is the generation of an intermediate process that can speak the protocol of all participating interfaces. A simple solution would have one protocol hand off to the intermediate process that in turn will hand-off to the next process. However, as discussed earlier, the optimal intermediate process would try to overlap the protocols as much as possible, resulting in an optimal exchange of data. Passerone's algorithm for incompatible interface synthesis will be discussed in the terms of the criteria established in the first section.

Algorithm Applicability and Extensions

Passerone's technique goes a long way to satisfying the criteria set forth in the first section. However, there are few items not addressed in his solution that are needed for tools that would perform component integration.

- A methodology is needed that will allow verification of point-to-point connections because interfaces will differ from component to component. Specification of data types is a logical choice.
- Dependent data signals must be considered in the analysis of protocols (e.g. the memory example given earlier).
- Also, the ability to broadcast a single data signal to multiple sources or to multiplex input to a device (e.g. memory arbitration) is desired.

Types

Types are introduced as a means of verifying compatibility between components and are assumed to be contiguous structures in memory. For this reason, types may be defined and composed using arrays, records, and primitive types. There is no limit to the extent of the definition.

Types are necessary to ensure proper connections between components, and to provide a formal means for describing what elements of data are transferred at different stages of a protocol's execution. Events are used to link a signal in the interface with the type being transferred. For example, when specifying signal participation in a protocol, events may reference any member of a type directly; also, it is possible to specify a range or a combination of members. In this way, the signals being used in the protocol are separate from the type being transferred between components.

Dependent Data Signals

Dependent data signals must be handled during the initial processing of interfaces. Deciding on a binding across protocol boundaries is difficult because these signals are not included in the dataflow yet they are elements of **K**. A trivial solution may be achieved by forcing designers to specify all data that must be retained across components inside the dataflow. However, there are circumstances where this is not ideal and it leads to cluttering. For example, specifying an address line on every component accessing memory is redundant information and should be removed from the dataflow for simplification. This leads to the second option. The data line and address line may be incorporated into the same type and considered to exist on the same signal. But, this method seems to deny the user of representing the component in its native form. Another alternative is to supply specific support for situations like memory access. It may be that very few protocols exhibit the properties of a memory interface and it can be considered a special case.

In the end, it may be clear that memory interfaces should be handled in a unique manner. However, because there is no clear solution, these signals will be connected using type definitions only. For each dependent data signal, there must be a unique type and a unique binding on the other interface. If a dependent data signal does not have a binding in the incompatible interface, then the connection of the two interfaces is illegal.

Broadcasting

In a dataflow process, data commonly splits and is processed on multiple paths. Hence, supporting this split in the path requires the ability to send data to more than one component. Ideally, when the dataflow is analyzed, interactions that have a single producer and multiple consumer relationship may be handled in the same way as a point-to-point relationship. Based on the current model, a single interface component will be generated to properly communicate data to all components simultaneously and in optimal time.

Multiplexing

In broadcasting data, all participants are guaranteed arrival of data because copies may be sent to each participating interface. However, multiplexing is a competition for access to a precious resource. As a result, proper integration requires analysis of the participating components and a schedule of access to be calculated. Multiplexing is needed in the case of memory or a comm-port. In both of these examples, independently acting components need access to the resource in order to proceed. If they do not receive timely access, the entire system may fail to meet specifications.

Multiplexing will not be considered further in this paper, but handling resource allocation and contention is important for system integration tools. Ideally, special user interaction should be avoided when synthesizing resource allocation tables and schedules. Based on the definition of interfaces and known behavior of components, a reasonable access scheme and a hardware interface should be generated to meet system specifications.

Implementation

An in depth discussion of the algorithm was originally published by Passerone et al [4]. This section will serve as a summary of the algorithm and the implementation.

The following concepts will be used to describe the implementation. Though similar to parts of the ACS environment, these do not have the same meaning.

- **Types** supported: primitives, arrays, and records. These are all user defined. Ideally, a large set of primitive types would exist and be naturally supported by the environment. Also, the types are assumed to be contiguous in memory because the data is moving through the network and must persist as a unit.
- **Signals** may either carry data or control logic. For every signal there must be an associated type, mode (e.g. input, output, or bi-directional), and initial value. The total set of signals corresponds to **D** in the interface description model. Port Signals are a subset of signals and correspond to **K**, the set of signals that must be retained across protocol boundaries.
- **Mealy Machines** are generated from regular expressions to represent the protocol via states, transitions, and events. Each interface (see below) contains one state machine.
- **States** are elements of the Finite State Machine. Each state may have multiple input/output transitions.
- **Transitions** connect exactly two states and contain a list of events segregated by direction (i.e. inputs or outputs).

- **Events** reference a signal and define its behavior on a transition. For example, an event may indicate receiving part of the data signal or the raising of a control signal. Events cannot measure passage of time or clock cycles, neither can they explicitly reference global system signals.
- **Interfaces** contain data signals, control signals, and a Mealy Machine model of the protocol using its signals. An interface with an empty Finite State Machine is allowed. Data is assumed available on every cycle. Interfaces correspond to the 3-tuple **(D,K,P)**.
- **Components** consist of one or more interfaces.
- **System** contains components and stores connections between components.

Initialization

Recall from the literature review the process of interface generation. Initially, there are connections from one component to one or more other components. While these connections represent the valid transfer of data, they do not describe how that transfer takes place. As a first step, an intermediate component is generated with a compatible interface for each participating component. At the system level, the current connections between components will be rerouted through the interface component. Also, the prior information regarding signal bindings will be recorded and kept for more setup.

The next step is to establish a record of signal bindings. From the provided dataflow, the known signal bindings are stored in hash tables. Note that the only signal bindings needed are for the set of signals in **K** because only these signals must be preserved. The hashing function maps a given Port Signal to a list of bound signals in order to support broadcasting and multiplexing functionality. Furthermore, there are hash tables inside the interface component to describe the relationship between input signals and output signals.

During processing of the product space, it will be necessary to know how much data of a given signal has been transferred to the interface process. A hash table will be used, for the duration of processing, to link signals with detailed information about their status. When a signal is considered, a hash table look up will be all that is needed to find or modify the information.

Finally, before processing can begin, the initial state and accepting state for the product state machine must be determined. The initial state is determined by taking the union of initial states from the Mealy Machines in each protocol the interface component. Likewise, the accepting state is calculated by taking the union of all accepting states. The computed initial state will serve as a starting point for the algorithm and the accepting state marks its termination.

Product Space Exploration

Currently inside the interface process, there are compatible interfaces for each participating component. Associated with these interfaces are hash tables defining the dataflow and a hash table storing information about each signal. At this point, the product space of the protocols associated with each interface will be analyzed to find the optimal solution for exchange of data between the components.

The algorithm used to examine the product space works as follows:

```
Explore() {
        Check if previously visited
        If data is consistent then {
                Loop through next states {
                        Explore next state
                        Record status for next state
                }
                Search through next state results and choose the best
        }
```

```
                Else set status to fail.
                Return determined status
        }
```

Processing begins with the initial state of each state machine. The Explore function is first called on the initial state. Because it is the first state processed, that state has not been visited and data must be consistent. Processing continues by calculating the set of next states for the product space. Next states are determined by permuting across the set of transitions leaving each current protocol state. Each transition has a single input and output state. Therefore, there will be a one-to-one mapping between next states and the set of exit transition permutations. For each next state calculated, the Explore function is called.

The first step of Explore asks the following questions.

- Is this state on the active state stack?
- Has this state previously failed?
- Has this state been accepted?

If any of these questions are true, the state has already been visited, or is in the process of being analyzed, then there is no need to continue analysis. If no analysis as been done, processing will continue by checking for data consistency.

Checking for data consistency errors allows for more efficient traversal of the product space. Data may become inconsistent via the following mechanisms.

- A receiving protocol falsely believes it has received data and attempts to process it.
- A sending protocol does not wait for the receiver to properly latch the data.

The state of data is determined by processing the events resident on transitions. When a next state is determined, the events stored on that transition are activated. For example, an event may record that a given data signal is available, or it may indicate that a component is reading that signal. Any signal affected by an event has its corresponding information updated to indicate the occurrence of that event. After all the events have been applied to the system, all affected signals and their bindings will be checked to ensure that anything sent out of the interface component had been previously received. If data is found to be inconsistent, the state is placed in a list of failed states and not traversed again. If data consistency is verified, each next state is calculated and the Explore function is called on each state.

The results of Explore are recorded by the state from which the search was called; either the search resulted in failure, a loop, or success. Of those paths that are accepted, the best is used as the return value for the state. In this way, the optimal path may be selected. The return value for the current state will be the best value it found. In the case of success, it adds one to the total distance to completion so that if comparisons between successful traversals are made, the path that minimizes latency may be selected.

VHDL Generation

VHDL code is generated from the resulting interface model to implement an interface component. A common generation template, Figure 4, is used for all generated interface components. The resulting component is centered around the generated Mealy state machine. Based on the current state, a mask is encoded that will inform the latch what input data is currently being received. As necessary, these values will be stored or forwarded directly through the process. Also, output masks are generated for each interface receiving data. This mask encodes for the appropriate combination of data that should be latched onto the set of output signals associated with that interface. There will be as many output multiplexors as there are components participating in the exchange. Finally, the input latch and output multiplexors are generated that will properly interpret the mask values. The VHDL is generated with the state machine and mask values in a behavioral format.

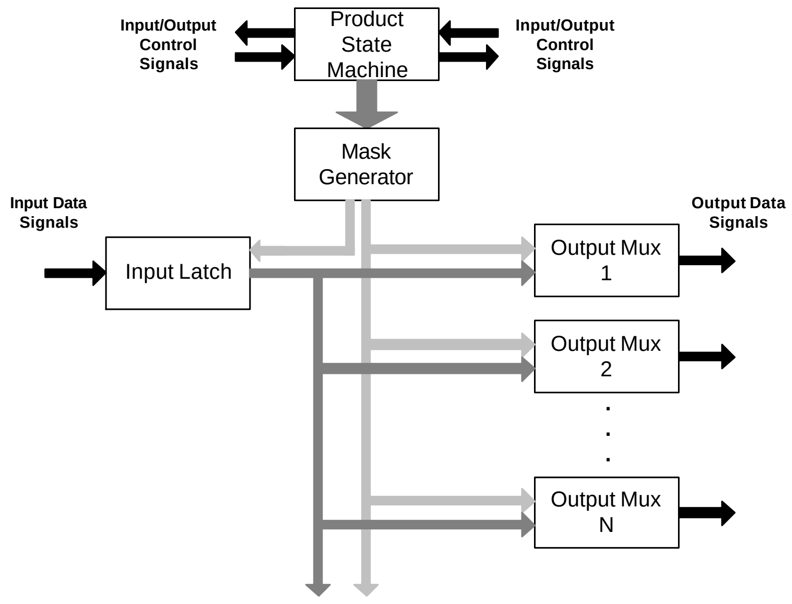


Figure 4: VHDL Generation Template

Analysis

The utility of incompatible interface synthesis can be studied by comparing it with the former method of system development. In Experiment I, several interfaces will be synthesized and implemented using the interface generation techniques. This will serve as a test of correctness for the proposed VHDL generation methodology, as well as an illustration of the capability of the interface generation techniques. Experiment II will focus on comparing systems that tend to use wrapping to achieve compatibility versus a system using generated interfaces.

Each experiment was performed on the Altera Flex 10k100ABC-356 (with a total of 4992 logic elements) using Synplicity's Synplify as the VHDL compiler. No optimized vendor-specific components were used in these implementations.

Experiment I

The first three entries in the table below show a least-significant-bit first protocol mapped to a most-significant-bit first protocol. These protocols represent worst case scenarios for the incompatible interface algorithm in terms of latency. The results for a 4-bit, 8-bit, and 16-bit implementation are recorded. The "States" column shown in the tables below records the minimal number of states for successful transfer. Because the data is transferred one bit at a time, it is reasonable to conclude that the protocol must have as many states as the minimal number of serial transfers. Because the last bit of the first protocol can be overlapped with the first bit of the second protocol, there are only seven steps to completion for the 4-bit implementation. The VHDL generation technique did not perform as well as a hard coded design. As bit width increased, so did the size and speed of the implementation. In a hand coded serial design, the speed should stay roughly the same because the combinational logic does not grow deeper with added width. The initial results suggest that serial components will perform poorly using this generation technique.

The next four rows show the results of a protocol communicating in parallel by raising a single ready line. This protocol is connected to another parallel protocol, a two-bytes, serial protocol, and a 16-bit, serial protocol. As expected, the serial protocol performs worse than the parallel or byte, serial protocols. However, combining all three receivers into a broadcast configuration yields an interesting

Table 1: Interface Synthesis Experiment I

Participating Protocols	Bits	States	Logic Elements	Clock Speed
LSB Serial to MSB Serial	4	7	26	59 MHz
LSB Serial to MSB Serial	8	15	59	53 MHz
LSB Serial to MSB Serial	16	31	187	30 MHz
Parallel to Parallel	16	1	19	54 MHz
Parallel to Two Byte Serial	16	2	28	51 MHz
Parallel to Serial	16	15	99	39 MHz
Parallel to Parallel,Serial,Two Byte	16	15	112	37 MHz

result. In this scenario, which includes a serial protocol, the clock speed as well as the logic resources are closely maintained to that of a parallel protocol communicating with a serial protocol. This suggests that the overlap in structure for generating multiple components is on the same order of efficiency as generating the parallel-to-serial transfer. One final note, the components performing significant, parallel operations tend to fair better than their serial counterparts.

Experiment II

The Serial-to-Serial Protocol that inversed bit order is the longest transaction shown above. This transaction is the closest resemblance to the effect of component wrapping, which often forces protocols to execute more in sequence than in step. Most of the other protocols which had more overlap, executed in fewer steps and in smaller space. Taking a component implemented using a wrapping mechanism and comparing it to the benefits of using interface synthesis techniques will illustrate any design, resource, or execution-time savings. Four systems were tested that implemented a 16-bit constant multiply piped into a 16-bit squarer. The core multipliers used to implement the components are shown in Table 2. Note that the size of components in Table 2 cannot be considered a direct function of bit width because some components are smarter than others. For example, the serial components have more logic to maintain state than do the parallel components.

Table 2: Core Component Implementation Details

Description	Bits	Logic Elements	Clock Speed
Constant Parallel Multiply	16	30	32 MHz
Constant Serial Multiply	16	70	39 MHz
Parallel Square	16	190	19 MHz
Serial Square	16	100	32 MHz

This basic connection was modified four different ways, allowing each component to take the role of a parallel or serial protocol. The permutations are shown in Table 3. Careful study of Table 1 and Table 3 reveal that synthesized solutions are very close to the sum of the core participating components plus the size of the generated interface. If the core components represent the optimal, basic components with which designers build systems, then a system which minimizes additional logic would be ideal. The results in Table 3 suggest that incompatible interface generation is an approach that cut downs on excess implementation.

The current VHDL generation techniques leave much to be desired. Consider the last two cases shown below; the result of synthesis yields no improvement over using wrapped components. On the other hand, the space savings of the first two tests were very significant. In the case of the first two tests in Table 3, eliminating the excess implementation in the wrapped components led to a drastic decrease in area. This type of code bloat is common when components are wrapped to achieve added functionality and compatibility.

In the last two tests in Table 3, there is no significant difference in size. Based on core component sizes, the synthesized Constant Serial Multiply mapped to a Serial Square might occupy roughly 170 logic elements plus the size of the interface. From Table 1, a Parallel to Serial interface may occupy 100 logic elements. This sums to around 270 logic elements, which is the approximate result in Synplify. In effect, the size of the synthesized interface is too large to achieve a gain in space or clock speed.

Table 3: Interface Synthesis Experiment II

Description	Style	Bits	States	Logic Elements	Clock Speed
Constant Parallel Multiply	Wrapped	16	8	501	19.5 MHz
to Parallel Square	Synthesized	16	2	231	18.4 MHz
Constant Parallel Multiply	Wrapped	16	23	537	19.5 MHz
to Serial Square	Synthesized	16	17	178	28.4 MHz
Constant Serial Multiply	Wrapped	16	24	299	21.4 MHz
to Parallel Square	Synthesized	16	17	303	20 MHz
Constant Serial Multiply	Wrapped	16	39	272	28.6 MHz
to Serial Square	Synthesized	16	17	273	22.1 MHz

When considering the data from Table 1, it is clear that largely parallel protocols tend to fair better in the selected VHDL generation scheme. Parallel protocols tend to require significant data buffering, but synthesis techniques can decrease the needed buffering. This results in large space savings in synthesis. However, serial components, in general, require less buffering and less logic per bit. Because the current synthesis template requires storing all the data for any transfer, synthesis performs poorly when serial interfaces are involved.

A possible solution is a modification to the VHDL code-generation techniques. It is reasonable that heuristics in the code generation process can improve code generation depending on a component's architecture. Instead of using a code generation template primarily benefiting parallel protocols, a template optimal for serial architectures may be selected. Data collected during processing may be used to select the appropriate template. For example, information may be tallied regarding the number of transfers and a minimal spanning set for holding incoming data can be determined. Based on the information collected, an interface component may be generated that stores and transfers the data most efficiently.

CONCLUSIONS

Component-based design provides a powerful abstraction for composing complex systems. In the ACSMIDE, enumerating the interfaces significantly aided the system in verifying construction. However, in practice, performance penalties arise when incompatible interfaces are 'wrapped' with system-supported protocols. To ameliorate the performance penalties due to integrating diverse components, a methodology was sought to represent interfaces in a generic manner in order to achieve optimal construction of complex systems.

As a solution, a model for describing interfaces was sought that would allow for compatibility checking and the generation of adapters between incompatible protocols. The number of signals, the type of information they carry, and how they are used in the protocol must all be captured to represent commonly used components in ACSMIDE as well as industry standard Intellectual Property. As a result, the interface description model was presented as a three-tuple, composed of a set of signals, a regular expression that defined the sequence of data transferred, and the set of required signals needed when connecting to another component. Using this model, techniques have been shown for generating compatible interfaces and incompatible interface adapters.

In the synthesis and implementation of incompatible protocols, an efficient resulting protocol is as important as an efficient VHDL implementation. The algorithm [4] presented for analysis of the product space produces an optimal sequence of events for transferring data between components. However, this does not guarantee an efficient implementation on an FPGA. The amount of data being transferred, the size of the transfers, and their sequence must be considered in order to achieve efficient implementations in terms of gate usage and potential clock frequency. In order to achieve optimal synthesized solutions, code generation techniques must use the knowledge collected in the analysis of the interface models to generate VHDL tailored for each interface.

As a test of current usability, the interface description model may be incorporated into the ACS Model Integrated Development Environment. Currently in ACSMIDE, protocols are specified as an attribute on each port of a structural primitive, and there are only a limited number of supported protocol types. However, it may be possible to specify a source file that would describe the protocol instead of choosing an item from a list. When a project is interpreted, the necessary files could be retrieved and analyzed, and an interface component would be generated. As a first step toward an integrated environment that would allow direct modeling of interfaces, a modeling paradigm may be developed based on the interface model described earlier. Models built in that environment would generate the source files used in the ACSMIDE.

Emphasizing functionality in component-based design marks a step in the direction of software first design. For systems specified on the functional level, it should be possible to reference components with similar functionality that operate with variable interfaces or on different architectures. Using a functional model as a guide [5], specific low-level parts that meet the functional specification may be selected then tightly integrated using generated interfaces. Eventually, an entire system may be synthesized on any available hardware from a concise description of the desired functionality.

REFERENCES

[1] Scott J., Bapty T., Neema S., Sztipanovits J. “Model-Integrated Environment for Adaptive Computing.” Proceedings of the Military and Aerospace Applications of Programmable Devices and Technologies Conference, CD-ROM Reference D5, Greenbelt, MA, September, 1998.

[2] Narayan S. and Gajski D. “ Interfacing Incompatible Protocols using Interface Process Generation.” Proceedings of the 32nd ACM/IEEE Design Automation Conference, 1995.

[3] Öberg J., Kumar A., Hemani A. “Scheduling of Outputs in Grammar-based Hardware Synthesis of Data Communication Protocols.” Proceedings of the Design Automation and Test in Europe, 1998.

[4] Passerone R., Sangiovanni-Vincentelli A., and Rowson J. “Automatic Synthesis of Interfaces between Incompatible Protocols.” Proceedings of the 35th Design Automation Conference, 1998.

[5] Hansen C., Uhlman M., and Rosenstiel W. “An Interface Description Model for Reuse of Algorithmic Hardware Specifications.” Forschungszentrum Informatk, Hardwarebeschreibungssprachen und Modellierungsparadigmen Workshop, Braunschweig, Deutschland, February 1999.