Automatic Synthesis of Interfaces between Incompatible Protocols

Roberto Passerone
Department of EECS
University of California at Berkeley
roby@eecs.berkeley.edu

James A. Rowson
Alta Group of Cadence
Sunnyvale, California
jimr@altagroup.com

Alberto Sangiovanni-Vincentelli
Department of EECS
University of California at Berkeley
alberto@eecs.berkeley.edu

Abstract

At the system level, reusable Intellectual Property (or IP) blocks can be represented abstractly as blocks that exchange messages. The concrete implementations of these IP blocks must exchange the messages through complex signaling protocols. Interfacing between IP that use different signaling protocols is a tedious and error prone design task. We propose using regular expression based protocol descriptions to show how to map the message onto a signaling protocol. Given two protocols, an algorithm is proposed to build an interface machine. We have implemented our algorithm in a program named PIG that synthesizes a Verilog implementation based on a regular expression protocol description.

1 Introduction

As time to market pressures and product complexities climb, the pressure to reuse complex building blocks (also known as Intellectual Property, or IP) also increases. Today most IP is available only at the RTL level. This is problematic because of verification speeds and the variety of signaling conventions used to for interfacing.

This paper addresses the problem of synthesizing interfaces between communicating IPs that use different signaling conventions. For this problem, a description of the entire behavior of the IP is not only cumbersome, but it introduces unnecessary details that may even hamper the design process. In [11] Interface-Based design was proposed as a methodology that attempts to separate the communication from the behavior for IP. To separate the communication, the blocks must be abstracted to a transaction or messaging level. With abstracted communication, improvement in simulation performance was shown with a simulator named Cheetah. However, the abstraction level that is appropriate for fast simulation is not efficient for implementation. By separating communication and behavior we can have a single abstract model that represents many concrete IPs that have different signaling conventions on their interfaces. For the purposes of this work, we are assuming that the concrete IPs have fixed signaling conventions and that in order to get two IPs to communicate, as specified at the abstract level, we will have to synthesize a machine to convert from one set of conventions (or protocol) to another.

Our problem formulation could then be expressed as: *Given two communicating design actors exchanging data, and a description of the two protocols that each one of them uses to transfer the data, determine an interface so that data transfers are consistent with both protocols*

A protocol may in general be given regardless of its physical implementation. However, we take a few simplifying assumptions on the implementation in order to limit the size of the design space. Besides, only the first two are really essential and they are often verified in practice.

1. The communication is *point-to-point*, that is the communication media (the interface we want to synthesize) is not shared with other modules in the system. The algorithm could however be used as a building block to a more general approach.

2. The two sides exchange data in the form of the same *data type*; more specifically, the data type is translated into a string of bits, therefore we require that the number of bits that are transmitted be the same as the number of bits that must be received.

3. During the transfer, part of the data will be temporarily stored in the interface: to simplify the solution of the problem we have considered only a very simple implementation of the storage, namely an internal register long enough to store the entire data type.

4. We assume that the modules willing to exchange the data *are driven by the same clock* and are therefore fully synchronous. However, we believe that the algorithm is applicable to asynchronous and event based systems with minor modifications.

As a final remark, it must be pointed out that in this first version of the algorithm we are only considering a single transaction; the extension to a stream of data requires the analysis of the interface storage allocation. Given the assumptions, the output of our algorithm is a *finite state machine* and a *data path* consisting of the internal register. Reading from and writing to the register can be done in a single clock cycle.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 98, June 15-19, 1998, San Francisco, CA USA
ISBN 1-58113-049-x/98/06…$5.00

As shown in Figure 1, the communication between an abstract producer and receiver must be implemented in the concrete world using an interface. We will synthesize this interface by using a declarative protocol description that relates the abstract communication to the signaling conventions on our concrete IP.

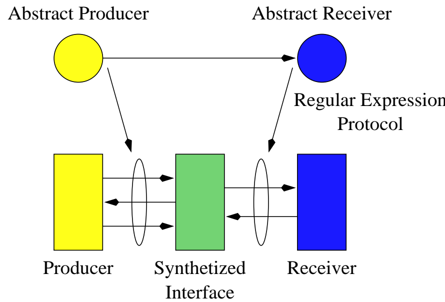

Figure 1: Problem formulation

This paper is organized as follows. First Section 2 gives a brief description of previous and related work; Section 3 describes the input language of our interface synthesis tool, while Section 4 presents the algorithm; some experiments and results are reported in Section 5 and finally Section 6 concludes the paper and presents some line of future work.

2 Related work

The problem of interface synthesis has been addressed in the literature. One of the first work is that of Borriello [3] [4], who introduces the "event graph" to establish the correct synchronization and data sequencing. The limitation of this approach is that the two protocols should be made compatible by manually assigning labels to the data on both sides, since the specification of the protocols is given at a very low level of abstraction using waveforms. Jane Sun [13] extends the approach by providing a library of components that frees the user from considering lower level details, but doesn't solve the problem we mentioned.

A different approach is that taken by Gajski et al. [8]: first, the protocol specification is reduced to the combination of five basic operations (data read/write, control read/write, time delay); the protocol description is then broken into blocks (called *relations*) whose execution is guarded by a condition on one of the control wires or by a time delay; finally the relations of the two protocols are matched into sets that transfer the same amount of data. Although this algorithm is able to account for data width mismatch between the two modules, the procedural specification of the protocols makes it difficult to adapt different data sequencing, so that only the synchronization problem is solved.

A third approach, similar to the one used in this work, is that of Akella and McMillan [2]: the protocols are described as 2 FSMs, while a third FSM represents the valid transfer of data. The product machine is taken and pruned of the invalid/useless states. The limitation here is that no data width mismatch can be handled and that the designer must manually enter the intended behaviour of the interface in the form of the third FSM (called the C-machine).

In our approach some of the mentioned limitations are overcome: the correspondence between pieces of data on the two sides is automatically resolved, the interface can translate between different sequencing of the data and no intended behavior must be introduced to describe the interface process. However, we assume that the two communicating parties are driven by the same clock, a limitation that is not present in [3] and in [8].

As we will see later, the two protocols are described using regular expressions. The use of regular expression for hardware description in general and for protocol description in particular is not new and has been already presented in the literature. The grammar-based specification employed in this work has been inspired mainly by the work of Seawright and Brewer [12] who demonstrated how effective regular expression could be for protocol and control intensive specifications. Hemani et al. also report in [9] the use of grammar-based specification for the synthesis of hardware for data communication protocols, although the specific problem of interface synthesis is not addressed.

The use of derivatives of regular expressions was introduces by Brzozowski [5]. The particular way in which the derivatives are computed in our work resembles that introduced by Coelho et al. in their work on *control-flow expressions* [6].

3 Protocol specification

As stated in the general problem formulation, the input to the algorithm is a description of the protocol used by the two modules. In our case, we assume that each module has a set of ports (data and control) over which the transfer occurs. We define a *protocol* as the *legal sequences of values that may appear on the ports from the onset to the end of the data transfer*.

If we order the ports in some arbitrary way, we can define a *symbol* in the protocol as a tuple composed of the values on the ports listed in their order. For simplicity of specification, ports that represent buses can be bundled together and assigned a single numerical value. Under this assumption, a protocol is simply a set of strings of symbols, or, in other words, a language in the alphabet of all the values that a symbol may assume. Our choice is to describe such a language with regular expressions: this way we can only use regular protocols, but a one to one correspondence can be easily established with finite state machines.

As mentioned before, symbols take values over the set of all possible tuples of values of the ports. If we included in the alphabet all the values that the data can take, then even very simple protocols would be expressed with exponentially growing regular expressions. For example, if a block has a connection to its environment with 8 wires transmitting a byte, then the protocol would be the set of all possible values over 8 wires, namely 256 different strings of 1 symbol. Since the interface is not concerned with the value that the data takes, but only with the control flow of the protocol, we introduce in the regular expression's alphabet a new symbol meaning *any value*. This choice is also unsatisfactory: in case of protocols where data is sequenced over a certain set of wires (e.g. a serial line), the interface must know what part of the data type is currently being transfered. Therefore we introduce a symbol with the meaning: *any value the data type takes on a certain subset of its string of bits*. For simplicity in parsing, we only allow the specification of such a *reference* over intervals (possibly a single bit) on the string of bits representing the data type. For example, if a data type `D` is composed of 20 bits, the syntax `D[10:5]` is a reference to the value of the data type from bit 5 to bit 10. The occurence of a reference in the protocol tells the interface either to store the value in the internal register or

to output the value previously stored in the internal register at the specified location.

A very simple programming language describes this information as a text file. To ease the protocol specification, the token can be thought of as a typical composite datatype, either an array or a record structure composing other types. Usually there is a rich set of primitive types available, although in the case of our work we limited the primitives to only one type representing a single bit.

The ports are simply listed with their direction and their data type (a bit or a set of bits). Since a protocol is independent of the direction of the transfer of data, the direction of the port is specified as either `master` or `slave`, the actual direction being resolved when the synthesis of the interface is requested. This way an input device and an output device that use the same protocol share the same description.

Regular expressions can be expressed hierarchically using a regular grammar [1]. In addition to its name, each rule of the grammar has a list of parameters whose value can be specified by the parent expression. In each rule, the references to values (as in `D[10:5]`) can only be expressed in terms of the rules parameters which act as local variables (very much like a function call). For the top level rule of the grammar, the only parameter is the token to be transfered so that, ultimately, all references are made with respect to it.

A symbol in the alphabet of the regular expression is a comma-separated list of values and references enclosed in braces, as in { `0, 4, D[10:5]` } for a 3 port specification. The number of values and their type must match that of the port list for all rules, regardless of their level in the hierarchy of the specification.

Symbols and regular expressions can be composed using any combination of the standard operators, including $*$ for Kleene closure (0 or more of the referenced symbol), $+$ for semi-closure (1 or more), | for choice and a comma for sequential expressions. Recursion is not allowed, except in the form of tail recursion when the Kleene closure operator is used.

The following two examples show how the token `yow` can be mapped to two different protocols:

```
type byte bit[7:0];
type yow { byte a; byte b; }

protocol serial of type yow {

 master bit start, byte bus;

 term null() { 0, - }
 term one(byte b) { 1, b }
 term two(byte b) { 0, b }

 serial(yow y) { null()*,one(y.a),two(y.b) }
}

protocol handshk of type yow {

 master bit trigger, byte bus

 term wait(bit t) { t, - }
 term get(bit t, byte b) { t, b }

 handshk(yow y) { wait(0)*,get(1,y.a)+,get(0,y.b)+ }
}
```

This first snippet of input code defines a protocol for type `yow` named `serial`. The interface being defined has two ports that can be used within the protocol, a pin named `start` and byte-wide bus named `bus`. Both of these ports are driven by the master side of the interface. There are three sub-rules defined in the grammar: `null`, `one`, and `two`. Each of these patterns must provide a value for all the ports, with don't care being represented by a dash. Parameters are listed with their data type after the name of the rule. The top level rule is a regular expression composition of calls to the terminal rules. In this case, `null` is expected 0 or more times (a Kleene closure), followed sequentially by a `one` with the `a` field of the `yow` token, followed by a `two` with the `b` field. As mentioned earlier, rules can be nested hierarchically so that `serial` could be used as a building block for more complex protocols. Our example `serial` protocol waits for a value of one on the `start` pin with an associated byte on `bus`, followed immediately by a zero on the `start` pin and the other byte on `bus`.

The second input code defines protocol `handshk` for type `yow`. Here the protocol starts with `trigger` with value 0. Byte `a` is transfered when `trigger` goes to a 1 and byte `b` is transferred when `trigger` goes back down to a 0. Notice the difference with the previous protocol: in this case the time spent on the first byte is not known, whereas in the previous case it was specified as a single clock cycle. In the first example the pin `start` only identifies the start of the transfer, in the second example the change of value in `trigger` identifies the transfer of the first *and* of the second byte.

4 Synthesis Algorithm

The goal of the synthesis algorithm is to obtain a finite state machine that when placed between the two modules implementing the specified protocols would make the communication possible. The problem is that of recognizing a given regular language on the producer side and to generate a proper string contained in the other regular language on the other side; the difficulty of the problem is maintaining the same "meaning" (i.e. preserving the data contents of the message) while trying to optimize some parameters (the transfer latency, the size of the storage in the interface process, etc.).

The approach that we will undertake is the following: first the regular expressions representing the two protocols are translated into two automata that recognize the corresponding regular language. These two automata form the bulk of the interface. Then the product of the two automata is taken so that only the legal sequence of operations is retained, the signals are translated into inputs and outputs, and the non-determinism that arises is resolved following one or more of the following rules:

1. never output a piece of data that has not yet been received
2. transfer the data
3. minimize (optimize) the latency

Any remaining non determinism is broken usin arbitrary choices based on the order the states are generated.

A simple example will help illustrate the construction of the product machine. Consider the two protocols described in section 3: their corresponding finite automata are shown in Figure 2. Suppose we wish to transfer data from the `handshk` to the `serial` protocol. The product machine construction starts from the state corresponding to the pair of initial states. Following that, all possible transitions are explored to see which ones should be included in the product.

Figure 3 shows the process as it evolves in the product machine exploration.

From the initial states (Figure 3.a) the set of all possible transitions includes a loop to the initial state itself or a transition to a pair of states where either one of the two automata has advanced one position. The pair of states shown on the right (Figure 3.b) is an instance of a forbidden transition, because we are trying to output the first byte that has not been received yet. That transition is therefore not included in the product machine. The transition on the left (Figure 3.c) can be safely taken since the first byte is received and nothing is produced.

From this state (and also from the initial state) we could move to the pair of states shown in the right transition (Figure 3.d): although this pair of states is legal, the serial protocol requires that the product machine move on the second shaded pair of states even when the second byte is not received, which is not legal. Since we have no control over this transition, then also the legal state should be considered illegal and the transition should not be taken. This condition cannot be detected immediately, and therefore it implies a backtrack in the algorithm. The alternative is therefore either to loop if the second byte is not received or to move forward with the left transition when it arrives. In the second case (Figure 3.e), the product machine allows a non deterministic choice: we can either transition to the first shaded pair or directly to the second shaded pair. Since our objective is to minimize the latency, the final product machine shown in Figure 4 moves directly to the second pair before reaching the final state that concludes the transaction.

The following paragraphs describe each step in detail.

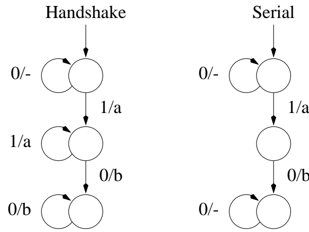

Figure 2: Finite automata

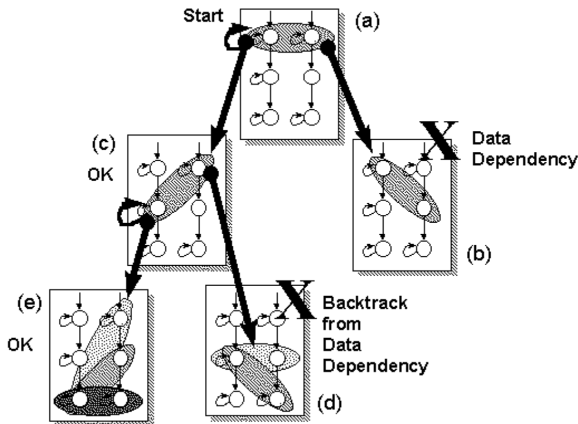

Figure 3: Product machine computation

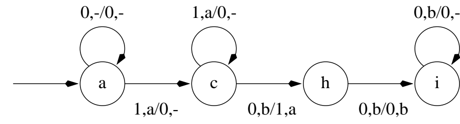

Figure 4: Final result

4.1 Automata generation

To translate the regular expression into a finite automaton we follow an approach based on the derivatives of regular expressions ([5]). A different algorithm would first build a non-deterministic automaton and then use the subsets construction to obtain the deterministic version (as described in [7]). The advantage of the derivative approach is that at each step we know how far we are in the protocol transaction, since we start constructing the automaton from the initial state rather than from some unspecified internal state. Exploiting this feature, we are able to characterize each state with the amount of data that has been transferred along the ports, an information that will be essential during the subsequent product machine construction.

As pointed out in [5], the challenge in computing the derivatives is recognizing whether two regular expressions (two derivatives) represent the same language, even though expressed in different form. In our algorithm, we represent the derivative of the regular expression with respect to a certain string as the set of symbols in the regular expression itself that may follow the given string (obtained by traversing a tree representing the regular expression using the operators as guides). Clearly, two identical sets of symbols represent the same derivative, thus the check for equivalence is reduced to a check of equality between sets; unfortunately the converse is not true, so that the worst case complexity of the algorithm is exponential in the number of symbols in the regular expression (the same as the subsets construction), and the finite automaton that is obtained may not be minimum. However, in all the practical cases that we have run through the algorithm, the state explosion was not observed: in fact this algorithm can easily factor any common term found at the beginning of the different branches of a choice operator, or detect the reconvergence after the choice was taken, thus following the style that designers naturally employ when specifying a protocol. More details can be found in [10].

4.2 Product Computation Algorithm

At the end of the previous step we have two deterministic finite automata that recognize a transaction on either side of the interface. The algorithm then proceeds with the construction of a subset of the product machine that performs the correct transfer of data. Moreover, the input/output nature of the signals is taken into account, so that a finite state machine rather than a finite automaton is eventually created.

An important issue that should be carefully examined is non determinism. The algorithm starts with two deterministic finite automata, so that the product is still a deterministic finite automaton. However, when signals are partitioned into inputs and outputs and a finite state machine is built, only the input set contributes to the condition on each arc, and therefore non determinism may arise (these machines are known as pseudo non deterministic, in that their corresponding finite automaton is deterministic). An example will help clarify this concept: if we consider a

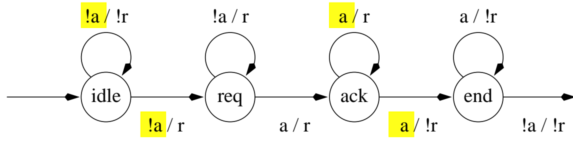

Figure 5: Non deterministic finite state machine

request-acknowledge protocol, the initial finite automaton would contain both the request and the acknowledge signals as input. The specification usually doesn't define the number of clock cycles intervening between the edges of these signals; when an FSM is built using this model, for example on the receiver side of the interface, the request signal is considered as an output and therefore the transition between an idle state and a request state, or between an acknowledge state and an end state (see Figure 5), becomes non deterministic. Obviously the final implementation must be deterministic, so that any choice must be taken at "compile" time. The nature and quality of these choices directly affect the quality of the interface in terms of correctness, latency, storage requirements and therefore performance. It is in fact the presence of non determinism that gives us the degrees of freedom necessary in the optimization process.

There can be basically two approaches to construct the required subset of the product machine: start from nothing and add states, or start from the entire product machine and remove the unwanted states. As shown in the simple example earlier in this section, in this implementation of the algorithm we follow the first approach (although a version using the reverse approach is under consideration). The algorithm systematically explores all paths going from the initial state to the final states, removing those that are not permissible. Considering the way non determinism is resolved, the following statement can be made:

Proposition 1 The Product Computation Algorithm always finds the correct interface between two protocols, if one exists. In addition, it always returns the minimum latency interface.

A more formal proof of this statement can be found in [10].

The algorithm that we use is a depth first recursive search implemented by a procedure called `explore`, outline in Figure 6. The process is started by the creation of the initial pair of states and a call to its `explore` function. Three data structures are used to support the computation: a *stack* records all the states that have been visited along the path to the current state, used to detect loops in the product machine; a *pool* records all the states that were found to be illegal either because of data inconsistency or because they lead to data inconsistency (this data structure acts like a cache that prevents the algorithm from re-exploring paths that are already known to be inconsistent); an *FSM* is used to collect all the states that will eventually be part of the product machine.

For each pair of states, i.e. a state in the product machine, a pair of bitsets is used to record the amount of data that has been received and that has been sent, or has to be sent. These values are updated each time a transition is taken between two pairs of states, and is used to check the data consistency.

The `explore` function returns an object to the caller which may assume different symbolic values: **Success** means that the state will certainly lead to a succesful transfer of the data and a companion number indicates the minimum

```
explore( ) {
  If on stack return ImmediateLoop
  If inconsistent return Fail
  If already in the FSM return previous result
  If previously explored and failed return Fail
  Push state on the stack
  For all pairs of outgoing transitions {
    Compute new state
    explore( new state )
  }
  Resolve non-determinism
  Compute return result
  Update data structures
  return
}
```

Figure 6: Function `explore`

number of clock cycles that it takes to get to the end of the transaction; **Fail** means that the state is either illegal or leads to an illegal state; **ImmediateLoop** indicates that the state has already been explored whereas **FutureLoop** that the state leads to a loop in one of its future transitions; **LoopSuccess** is returned when the state may either lead to a loop or to a succesful transfer of the data (depending on the behavior of the outside environment). In order to compute the return value, `explore` starts by checking if the state was already explored and returns immediately to avoid multiple computations. Otherwise the state is pushed on the stack and for each pair of outgoing transitions the new state is created and explored (after updating the data consistency bitsets). The return value of each exploration is stored and the state is popped from the stack.

Then the set of outgoing transitions is partitioned into equivalence classes: each class is denoted by the same input label (i.e., the non deterministic transitions are grouped together). For each equivalence class, only one transition is chosen to be part of the final implementation. Here is where choices can be taken to resolve non determinism and to optimize performance. In particular, a transition whose exploration returned a **Success** or **LoopSuccess** is always chosen against a **Loop** or a **Fail**. Ties between successful transitions are broken considering the number of clock cycles to the end of the transaction.

The final result is computed considering the exploration result from all the transitions that survived the determinization. Since now all transitions are deterministic, a **Fail** in any of them will make `explore` return a **Fail** to the calling function even though other transitions from the same state lead to a successful transaction. This is because we now have no control over the transition that is taken, therefore correctness requires us never to reach the state. If there is no **Fail**, then either **Success** or **LoopSuccess** is returned with a number of clock cycle equal to the minimum over all transitions plus 1 (to account for the present state).

Before returning, if successful, the state is recorded in the FSM data structure along with its transitions. If unsuccessful, it is inserted in the failing pool.

When the last `explore` returns the FSM contains the product machine. Since states are added to the FSM starting from the last state (because of the recursive call), we may add states that are unreachable from the start state. A final clean up procedure takes care of removing all dangling states.

From - To	States	Rec.	Full Rec.
handshake - serial	4	14	5
serial - handshake	3	13	5
atm ser. - atm req-ack	1438	2983	1490
atm req-ack - atm ser.	111	3416	1680
atm ser. rev - atm req-ack	116	235	116

Table 1: Experimental results

5 Experimental results

The algorithm has been implemented in about 4500 lines of Java code in a program called PIG. PIG is able to load the textual description of the protocols, parse them, create the finite automata with the derivatives approach and produce the subset of the product machine as previously outlined. In addition to that, the Verilog code in the form of a finite state machine can be generated to be simulated with the corresponding models of the producer and the consumer.

A few examples have been run through the entire process. Table 1 shows a summary of some of the performance data. In particular we are interested in the number of states that are visited during the product machine construction. The columns *Recursion* and *Full Rec.* show the number of times that the `explore` function was called during a computation, and the number of times that the explored state was not found in any of the auxiliary data structures. Although the number of explorations can be more than the total number of states in the product machine, full exploration is necessary only for far less states.

Example `atm` is a large case exapmple where a serial synchronous protocol was interfaced to a 4 phase request-acknowledge protocol and vice versa for a packet consisting of 56 bytes. The serial side transfers data 8 bits at a time, while the request-acknoledge uses a 32 bit bus. Both protocols result in a finite automaton with 57 states each. In all cases PIG produces the correct result and the latency is minimized. The first case contains many states because the output of the data is concurrent with the input of the data in the interface. On the other hand, in the second case the interface needs to wait until all the data from the request-acknoledge protocol has been received (because it must be output serially without interruption), and only then can the output start: in this case less choice means less states, but also a higher latency. As an experiment, we've tried reversing the order of the data of the serial protocol in a serial to req-ack interface: in this case, too, the interface has to wait till the end of the input phase before starting the output.

Notice that although the number of states in the FSM is very different for the first two atm examples, the number of full explorations is about the same. It's also very intersting to notice in the last example how the huge number of illegal states and the use of the failing pool requires us to visit only a very limited number of states in the product machine, thus obtaining the result really fast.

6 Conclusions and future work

The experimental results show that the proposed algorithm is a viable tool to synthesize an FSM to be used as an interface between two protocols. Still, there are a lot of limitations to be overcome and a lot of interesting extensions. Some of the assumption could be relaxed: in particular we are looking into ways to extend the methodology to asynchronous protocols, and to let many parties get involved in the communication. Moreover, an analysis of the product machine should let us easily factor it into two distinct machines (since that is where we started from) and trade-off the number of states with the number of *conditions* that must be setup between *anchor points*. The problem of a stream of data versus a single transaction should also be addressed.

An interesting extension would be the use of this methodology to create an interface with modules that are implemented in software in embedded systems. In that case, the references to values could be replaced by functions that know how to fetch the corresponding data.

7 Acknowledgments

The authors would like to thank Ellen Sentovich, Luciano Lavagno, Gaetano Borriello and Luca Carloni for many suggestions and discussions.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1988.

[2] J. Akella and K. McMillan. Synthesizing converters between finite state protocols. In *Proceedings of the International Conference on Computer Design*, pages 410–413, Cambridge, MA, October 14 - 15 1991.

[3] G. Borriello. *A New Interface Specification Methodology and its Applications to Transducer Synthesis.* PhD thesis, University of California at Berkeley, Berkeley CA, 1988.

[4] G. Borriello and R. H. Katz. Synthesis and optimization of interface transducer logic. In *Proceedings of the International Conference on Conputer Aided Design*, November 1987.

[5] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the Association for Computing Machinery*, 11(4):481–494, October 1964.

[6] C. N. Coelho and G. D. Micheli. Analysis and synthesis of concurrent digital circuits using control-flow expressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):854–876, August 1996.

[7] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1986.

[8] S. Narayan and D. D. Gajski. Interfacing incompatible protocols using interface process generation. In *Proceedings of the* 32nd *Design Automation Conference*, pages 468–473, San Francisco, CA, June 12 - 16 1995.

[9] J. Oberg, A. Kumar, and A. Hemani. Grammar-based hardware synthesis of data communication protocols. In *Proceedings of the* 9th *International Symposium on System Synthesis*, pages 14–19, La Jolla, CA, November 6 - 8 1996.

[10] R. Passerone. *Automatic Synthesis of Interfaces between Incompatible Protocols.* M.S. Thesis, University of California at Berkeley, 1997.

[11] J. A. Rowson and A. L. Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the* 34th *Design Automation Conference*, pages 178–183, Anaheim, CA, June 9 - 13 1997.

[12] A. Seawright and F. Brewer. Clairvoyant: A synthesis system for production-based specification. *IEEE Transactions on VLSI Systems*, 2:172–185, June 1994.

[13] J. S. Sun and R. W. Brodersen. Design of system interface modules. In *Proceedings of International Conference on Computer Aided Design*, pages 478–481, 1992.