High-Level Specification and Automatic Generation of IP Interface Monitors *

Marcio T. Oliveira
Department of Computer Science
University of British Columbia
oliveira@cs.ubc.ca

Alan J. Hu
Department of Computer Science
University of British Columbia
ajh@cs.ubc.ca

ABSTRACT

A central problem in functional verification is to check that a circuit block is producing correct outputs while enforcing that the environment is providing legal inputs. To attack this problem, several researchers have proposed monitor-based methodologies, which offer many benefits. This paper presents a novel, high-level specification style for these monitors, along with a linear-size, linear-time translation algorithm into monitor circuits. The specification style naturally fits the complex, but well-specified interfaces used between IP blocks in systems-on-chip. To demonstrate the advantage of our specification style, we have specified monitors for various versions of the Sonics OCP protocol as well as the AMBA AHB protocol, and have developed a prototype tool that automatically translates specifications into Verilog or VHDL monitor circuits.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer Level Implementation**]: Design Aids; B.6.3 [**Logic Design**]: Design Aids; C.0 [**Computer Systems Organization**]: General—*Systems specification methodology*; J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD)

General Terms

Documentation, Languages, Verification

Keywords

Formal Verification, Regular Expressions, Pipelining, Alternation

1. INTRODUCTION

Standard design practice is block-based — the design task is carved into small pieces to be tackled by an individual designer or a small team. In the past, block boundaries and interfaces have been casually negotiated face-to-face among the designers. This informal negotiation does not scale with the push for higher productivity and complexity. In addition, we would like to reuse pre-designed and pre-verified IP blocks — either designed previously in-house or purchased from third-party IP suppliers. As a result, the trend is towards designing with large, complex blocks with well-defined functionality and interfaces.

This trend generates two complementary verification problems: how to verify that a block behaves properly *in its intended environment* without having to model and verify the rest of the system, and how to verify that a system behaves properly without having to instantiate all blocks and flatten the design. Current verification practice for the first problem is to create by hand an abstracted environment model for formal verification or a testbench for simulation-based verification. Current practice for the second problem is to create by hand abstract models of the blocks and the system, or else to attempt to verify the whole system and suffer from state explosion in formal verification or slow simulation speeds and poor coverage during system-level simulation. In either case, this practice is labor-intensive, error-prone, and results in time-consuming false error reports if the models are too flexible or missed bugs if the models are too strict.

Several groups have proposed interface-monitor-based methodologies (e.g., [8, 13, 7]) to address this problem. The common theme is to create a monitor circuit that watches the interface between a block and the rest of the system and flags any violations of the interface protocol (Figure 1). The key insight, empirically confirmed in several case studies, is that designing a passive, declarative monitor is easier than designing an active stub to model the environment. Furthermore, because the monitor is symmetric between the block and the rest of the system, the same monitor can be used to verify both the block with the system abstracted as well as the system with the blocks abstracted, thereby supporting a compositional/hierarchical verification style. The monitor also provides a precise documentation of the interface, on which formal sanity checks can be applied. Finally, it is possible to convert a monitor circuit automatically into a testbench (stimulus generator) for simulation-based verification [15]. The advantages of a monitor-based methodology are compelling.

Unfortunately, although impressive monitors have been built [12], creating a monitor for a complex protocol is a challenging task. This paper introduces a high-level specification style designed explicitly to simplify specification of interface monitors. Our goal is to provide an extremely easy way to generate monitors for common interface idioms. With numerous emerging standards for system-on-chip interconnect, the need for a simple, concise, and readable way to specify interface protocols is clear. Being able to translate these high-level specifications automatically into monitor circuits allows tapping the power of monitor-based methodologies. By using our specification style, IP suppliers will be able to formally verify that their cores conform to an interface protocol as well as supply a monitor for that protocol that is both easily human-readable and directly usable by verification tools.

1.1 Related Work

Verifying a block with respect to an environment has been a concern since the beginnings of hardware verification. CTL Model Checking [6], for example, assumes a closed system and implicitly requires writing environment models. Theoretical results show that modifying CTL model checking to explicitly handle environmental inputs (rather than closing the system with an environment model) imposes a large complexity cost [9]. For space reasons, we discuss here only the two lines of research that are the immediate parents

*This work was supported in part by an NSERC research grant and the MITACS Network of Centres of Excellence.

DAC 2002, June 10–14, 2002, New Orleans, Louisiana, USA.

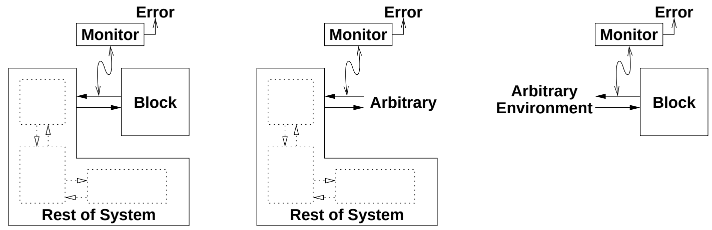


Figure 1: A monitor circuit watches the interface and flags any violations of the protocol. The block and system can be formally verified separately. The monitor can also be converted into a simulation testbench.

of the current work: interface monitors and regular expressions.

Using monitor circuits to encapsulate properties to be checked is an established idea. Indeed, a practical, industrial-strength verification methodology can be built on extensive use of monitors [4]. Our work was directly motivated by the elegance and power of monitor-based approaches to interface specification [8, 13, 7]. The emphasis of those works was mainly on the value of this way of thinking; little emphasis is on the specification language. Our focus is on the specification language; we seek to harness those results by providing a shortcut to specifying monitors.

The other parent of our work is the use of extended regular expressions to generate finite-state machines. For property specification, both Intel's ForSpec[2] and IBM's Sugar[3] include regular expressions and have ardent adherents. The most direct influence on the present work, however, is earlier work from the synthesis community: Production-Based Specification [11]. This work uses an extended regular expression language to specify state machines, which are synthesized in polynomial time into circuits, never explicitly building a deterministic finite-state machine and thereby avoiding a potential blowup. Production-based specification has proven to be particularly well-suited to synthesizing protocol state machines, and hence was a natural starting point for our research.

1.2 Contributions

The most obvious contribution of our research is the demonstration that regular expressions work very well for specifying IP interface monitors. That statement, however, is actually false. Existing specification styles based on regular expressions do rather poorly as soon as the interface protocol becomes as complex as typical system-on-chip interconnect protocols. However, by introducing two novel extensions — storage variables and a pipelining operator — we have created a specification style that does work very well for interface monitors. The new extensions require a new algorithm for translating specifications into monitor circuits. We have implemented this algorithm in a prototype tool that translates specifications into monitor circuits in Verilog or VHDL. Finally, we have demonstrated the usefulness of our specification style by developing monitors for two standards for system-on-chip interconnect: large portions of ARM's AMBA AHB high-performance bus protocol [1] and several versions of OCP (Open Core Protocol) originated by Sonics [14].

2. SPECIFICATION STYLE

We introduce the specification style incrementally, starting with regular expressions, and then introducing productions, storage variables, and pipelining. Examples taken from the AMBA AHB protocol will illustrate the concepts. We will try to provide enough information for readers unfamiliar with AHB to understand the examples. The complete AHB models are too large to include in a paper, but we will present a complete master monitor for a simple variant of OCP in the next section.

Fundamentally, a regular expression specifies a language, which is just a set of strings, which are sequences of characters, which are drawn from some alphabet. Analogously, we start at the very beginning with the alphabet of our specification style. The interface between a block and the rest of the system consists of a bunch of wires: some are inputs to the block; some are outputs. For example, an AHB slave device interfaces to the system via several wires, such as `HADDR[31:0]`, a 32-bit address input; `HRDATA[31:0]`, a 32-bit data output; `HWRITE`, `HTRANS[1:0]`, `HSIZE[2:0]`, `HBURST[2:0]`, which are control signals describing the type and size of a transfer; and `HREADY`, `HREADYOUT`, and `HRESP[1:0]`, which are hand-shaking and response signals. All of these wires are inputs to the monitor, which passively watches their values. Accordingly, the fundamental building-block of our specifications is an assignment of values to the wires on the interface at a given clock cycle.

For convenience, we allow the user to specify any Boolean formula on the interface wires. For example, the AHB protocol defines encodings for the different transfer and response types, so we allow the user to specify:

```
define idle   = !HTRANS[0] & !HTRANS[1];
define busy   =  HTRANS[0] & !HTRANS[1];
define nonseq = !HTRANS[0] &  HTRANS[1];
define seq    =  HTRANS[0] &  HTRANS[1];

define okay  = !HRESP[0] & !HRESP[1];
define error =  HRESP[0] & !HRESP[1];
define retry = !HRESP[0] &  HRESP[1];
define split =  HRESP[0] &  HRESP[1];
```

Any Boolean formula on the interface wires (and `defined` formulas) is a *primitive expression*.

Given primitive expressions, we can define regular expressions recursively in the usual manner. Any primitive expression is a regular expression. A regular expression concatenated to another regular expression is a regular expression. We use a comma as our concatenation operator. For example, the AHB specification defines different response codes for a slave to signal to the master:

```
wait_state -> !HREADY & okay;
okay_resp  ->  HREADY & okay;
error_resp -> (!HREADY & error) , (HREADY & error);
retry_resp -> (!HREADY & retry) , (HREADY & retry);
split_resp -> (!HREADY & split) , (HREADY & split);
```

The error, retry, and split responses take two cycles: the first with `HREADY` low, the second with `HREADY` high. The choice (denoted by `||`) between regular expressions is a regular expression. For example, to specify that a transfer can be one of four different kinds, we can write:

```
transfer -> idle_trans || busy_trans ||
            nonseq_trans || seq_trans;
```

Finally, we have the Kleene closure to denote repetition:

```
resp -> wait_state* , (okay_resp ||
        error_resp || split_resp || retry_resp);
```

The above expression specifies the response phase to be any number of wait states, followed by one of the response types.

For notational convenience, we use *productions* as they were defined in production-based specifications [11]. We have actually been using productions already in the preceding paragraph. The symbol to the left of the `->` operator is defined to be an abbreviation for the regular expression on the right-hand side. To guarantee that specifications correspond to finite-state machines, productions cannot be recursive.

The above definitions are the same as earlier regular-expression specification styles and appear to be convenient for describing protocols. The behavior of an AHB slave device, for example, is simply a sequence of transfers or idle periods: `slave -> (slave_idle || transfer)*` where `slave_idle` means `HREADY` is low or the slave is not selected, and `transfer` is defined as above. Describing the full details of typical IP block interface protocols, however, quickly reveals the limitations of a pure regular-expression specification style.

The first major obstacle is persistent storage of information. In AHB, for example, a slave device can reply with a *split* response, indicating that it needs a long time to complete the request. The interface monitor for a split-capable slave should remember the ID numbers of all masters who have splits pending, to ensure that a slave does not signal completion of a split transaction that has not happened. Encoding such information with regular expressions is possible, but painful: for *every possible value of the saved information*, the user must write a slightly modified version of every production that is affected. Instead, we propose *storage variables* as a simple alternative. The user can declare finite-state variables as part of the specification. At any point in a regular expression, values can be assigned to the storage variables. The values of the storage variables are available in any Boolean formula defining a primitive expression. The monitor for an AHB slave could have a 16-bit storage variable, with one bit for each possible master to indicate whether it has had a request that has been split. Whenever the slave issues a split, the corresponding bit is set; whenever the slave issues a split completion, the corresponding bit is checked.

The other major obstacle is pipelining. Almost all high-performance interfaces are pipelined to some degree. Most specifications describe the cycle-by-cycle behavior of an interface, but unfortunately, pipelining is extremely hard to specify (or understand) at the cycle-by-cycle level. Trying to specify pipelining via regular expressions or any other cycle-by-cycle style requires the user to entangle all the possible parallel behaviors by hand, resulting in a difficult, error-prone specification process and an unreadable specification. Instead, pipelining is most naturally understood as an operation that overlaps sequential operations (Figure 2). In the AHB protocol, the arbitration phase, address (request) phase, and data (response) phases are all pipelined. The official AMBA specification document [1] describes these phases sequentially in English, and then presents timing diagrams to attempt to show how they entangle in pipelined operation. Our solution is to provide an explicit pipelining operator, similar to the concatenation operator. For example, for an AHB slave monitor, a transfer has an address phase followed by a response phase:

```
idle_trans -> (idle & HSEL & HREADY) , okay_resp;
busy_trans -> (busy & HSEL & HREADY) , okay_resp;
nonseq_trans -> (nonseq & HSEL & HREADY) , resp;
seq_trans -> (seq & HSEL & HREADY) , resp;
```

(`HSEL` indicates this slave is selected; `HREADY` is the handshake that indicates the address phase is complete.) However, the address and response phases are pipelined, so that the response phase of one transfer occurs at the same time as the address phase of the next transfer. In our specification style, we simply replace the concatenation operator with the pipeline operator `@`:

```
idle_trans -> (idle & HSEL & HREADY) @ okay_resp;
busy_trans -> (busy & HSEL & HREADY) @ okay_resp;
nonseq_trans -> (nonseq & HSEL & HREADY) @ resp;
seq_trans -> (seq & HSEL & HREADY) @ resp;
```

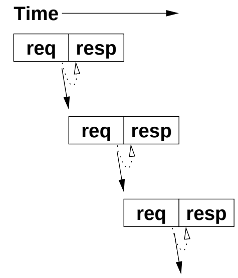


Figure 2: This figure shows multiple pipelined transactions, where each transaction has a request phase and a response phase: `(req@resp)*`. Our pipeline operator marks the point where the next computation overlaps the current one. At that point, we fork a new "thread" to complete the current transaction (dotted arrow), while the current thread continues with the rest of the regular expression, if any (solid arrow).

The semantics of the pipeline operator are that the thread of control forks into two sub-threads when the pipeline operator is encountered: one sub-thread continues with the regular expression as if the right-hand operand of the pipeline operator did not exist, the other sub-thread focuses only on the right-hand operand, ignoring the rest of the regular expression. The thread accepts a string only if both sub-threads accept (Figure 2). Multistage pipelines are easily specified as `(a @ (b @ (c @ ...)))`.

The ability to generate multiple threads where all threads must accept (pipelining) or at least one thread must accept (choice) resembles alternating automata, which in general require a double-exponential state-space blow-up to convert into ordinary automata [5]. To guarantee efficient translation into monitor circuits, we impose a few restrictions. First, we require the expression contained within a Kleene star not to accept the empty string. Known constructions can normalize regular expressions to obey this restriction [10], but our implementation does not currently include this step. Second, we forbid non-deterministic choice: we allow the choice operator, but the choices must be distinguishable within the first clock cycle. In practice, this restriction is not a problem because protocols are typically designed to make it easy to determine immediately what action is occurring. Finally, we allow at most one thread at a time to execute in a pipeline stage. For example, the expression `(a@(b,c))*` generates an error when the second repetition arrives at the `b` while the first repetition's pipeline sub-thread is still at the `c`. This restriction corresponds to allowing only one transaction at a time to use the hardware resources devoted to a pipeline stage.

We close this section with an example of the clarity of our specification. The official AMBA AHB specifications require a two-cycle response for a retry (shown above in the production for `retry_resp`), but also require an immediate OK response to a busy transfer. In August 2001, the question arose in the `comp.lang.verilog` and `comp.sys.arm` newsgroups of what happens if a slave starts to issue a retry (to the previous transfer) in the same cycle that the master issues a busy transfer, e.g.:

Clock Cycle:	1	2	3
Request:	Sequential	Busy	
Response:		Retry1	Retry2 or OK??

The sequential transfer at cycle 1 needs a two-cycle retry response

```
/* Inputs and Outputs of Master */
/* The monitor treats these as inputs. */
input SCmdAccept, SResp[1:0], SData[31:0];
output MAddr[31:0], MCmd[2:0], MData[31:0];

/* Response codes defined in standard. */
/* NULL, Data VAlid, and ERRor */
define null_resp = !SResp[0] & !SResp[1];
define dva_resp  =  SResp[0] & !SResp[1];
define err_resp  =  SResp[0] &  SResp[1];

/* Commands defined in standard. */
define cmd_idle  = !MCmd[0] & !MCmd[1] & !MCmd[2];
define cmd_write =  MCmd[0] & !MCmd[1] & !MCmd[2];
define cmd_read  = !MCmd[0] &  MCmd[1] & !MCmd[2];

master -> (cmd_idle || transfer)*;

transfer -> write_transfer || read_transfer;

/* SCmdAccept indicates that the slave
   has accepted the command. */

write_transfer ->
    (cmd_write & !SCmdAccept)*, (cmd_write & SCmdAccept);

read_transfer ->
    (cmd_read & !SCmdAccept)* ,
    (wait_state_resp || instant_resp);

wait_state_resp ->
    (cmd_read & SCmdAccept & null_resp) ,
    null_resp* , response;

instant_resp -> (cmd_read & SCmdAccept & dva_resp) ||
                (cmd_read & SCmdAccept & err_resp);

response -> dva_resp || err_resp;
```

Figure 3: A complete interface monitor for a Basic OCP master that can only handle one outstanding transaction at a time.

during cycles 2 and 3, but a busy transfer at cycle 2 demands an immediate OK response at cycle 3. Because of the pipelining, the questioner was unsure what would happen. Of the several answers offered in the newsgroup, most were wrong. In our specification, the `seq_trans` and `nonseq_trans` productions have a pipelined `resp` phase, which means that one sub-thread enforces that the response phase behaves correctly while another sub-thread continues to the next transaction. For a retry response, the `retry_resp` production shows that `HREADY` must be low in cycle 2, whereas the `busy_trans` production shows that `HREADY` must be high in cycle 2. Since the slave controls `HREADY`, this means that the busy transfer cannot occur at cycle 2, regardless of what the master requests. The only legal behavior is for the request at cycle 2 to be ignored (`slave_idle` in the productions above), and for the second cycle of the retry response to occur at cycle 3.

3. EXAMPLE: OCP

In the preceding section, we actually saw most of the specification for an AMBA AHB slave monitor. The AHB master monitor is too complex to present here, but for the sake of completeness, we would like to present a complete monitor for the interface between a master device and the rest of a system. Figure 3 shows the entire monitor for a master in a simple version of the Open Core Protocol (OCP) [14]. OCP is actually a very broad, parameterized family of protocols, spanning an enormous range of performance and cost objectives. The interface monitor shown is for a Basic OCP master with only one transaction in-flight at a time.

In Basic OCP, the master presents a command at the same time as the address, as well as the data if the command is a write. These values must be held constant until the slave accepts the command by asserting `SCmdAccept`, which could happen in the same cycle that the command is presented. Writes are posted (no response required once the slave accepts the command), but read commands have a response phase during which the slave sends data back to the master. The slave can insert zero or more wait states by keeping `SResp` set to the null response. The slave terminates the response phase by setting `SResp` to indicate that the data is valid or an error occurred. `SResp` can go non-null in the same cycle as `SCmdAccept` is asserted, which can be in the same cycle as the master presents a command.

Our specification in Figure 3 starts by declaring the interface wires, and then defining the command and response encodings. The first production defines the behavior of the monitor — in this case, we declare that the master's interface should exhibit a sequence of idle commands or transfers. A transfer can be a write or a read. A write transfer consists of zero or more states waiting for the slave to accept the command, followed by the slave's accepting the write. The read transfer is a bit more complicated, starting with zero or more states waiting for the slave to accept the command, followed by either an instantaneous response or a response with zero or more wait states.

For simplicity, the specification in Figure 3 does not check that the master holds the address and data values constant if the slave does not accept the command immediately. To enforce this requirement, we need only make a few changes to the specification. First, we would declare some storage variables to remember the values of the address and data:

```
internal hold_addr[31:0] = 0;
internal hold_data[31:0] = 0;
```

Next, we modify the transfers so that if the slave does not accept the command immediately, we remember the address (and data if applicable):

```
write_transfer ->
    (cmd_write & SCmdAccept) /* Same cycle accept */
    ||
    (
     (cmd_write & !SCmdAccept)
        /* Store original address and data. */
        { hold_addr <- MAddr; hold_data <- MData; } ,
     (cmd_write & !SCmdAccept &
        (hold_addr == MAddr) & (hold_data == MData)
     ) * ,
     (cmd_write & SCmdAccept &
        (hold_addr == MAddr) & (hold_data == MData))
    );
```

The read transfer production is modified similarly. To enforce the same constraints using regular expressions without storage variables would require separate read and write productions for each possible value of the address and data.

4. TRANSLATION INTO CIRCUITS

The translation process starts by macro-expanding all productions, since the productions cannot be recursive, resulting in a single (extended) regular expression for the monitor. In theory, this expansion can produce an exponential size blow-up, but in practice, this is often not a problem. The translation from an extended regular expression to circuits can best be understood as recursively building a circuit for each sub-expression, so the structure of the circuit exactly matches the structure of the regular expression. The circuit passes activation signals from sub-circuit to sub-circuit, corresponding to possible parses of the input string by the regular expression. We will elaborate on this construction below.

Our translation is similar to previous work in efficiently converting regular expressions into circuits [11, 10]. The key differences of our algorithm are building a monitor circuit, rather than a recognizer circuit, handling storage variables, and handling pipelining.

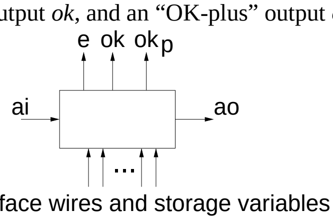
The first difference is that we are interested in monitoring the on-going behavior of an interface, rather than recognizing a regular language, which was the focus of previous work. A recognizer asserts its "OK" output only when the input sequence is a string in the language of the regular expression. A monitor, on the other hand, asserts its OK output as long as the sequence seen so far has not done anything not permitted by the regular expression. Accordingly, our logic that tracks the correspondence between the interface and the regular expression (the activation signals) is essentially the same as previous work, but the logic to generate the OK signal is completely different (described below).

Storage variables are simply registers in the circuit, whose output are available to the logic generated for primitive expressions. Assignments to storage variables are enabled by the same activation signals that track the parsing of the input string.

Pipelining is the most difficult difference. Intuitively, we will create a single thread for each pipeline stage, and the circuit for each thread behaves roughly like previous translations of regular expressions into circuits. The monitor is satisfied only if all active threads are satisfied. Additional bookkeeping is required to track the exact status of each thread.

More precisely, take the (macro-expanded) parse tree for the monitor's regular expression and delete the right-hand-operand edges of all pipeline operators, resulting in several, disjoint parse (sub-)trees. Our restrictions on the specifications (deterministic choice and single thread per pipeline stage) guarantee that each sub-tree will support exactly one thread. Each thread i will generate a thread enable output $tenable_i$ and a thread OK output tok_i. The monitor is satisfied as long as for all threads, $tenable_i \Rightarrow tok_i$. (We use $\Rightarrow$ to denote logical implication.)

Each regular (sub-)expression is converted into a circuit that can read all storage variables and interface wires. The circuit also has an activate-in input a_i, an activate-out output a_o, a circuit-enabled output e, an OK output ok, and an "OK-plus" output ok_p:



Intuitively, the activate signals indicate where a thread is in the regular expression, the enabled signal e indicates if this sub-circuit is enabled (is trying to match the interface signals), and the OK signal indicates that the sub-circuit is enabled and agrees with the current values on the interface wires. The OK-plus signal is a technical detail needed to handle the possibility of recognizing the empty string with a Kleene star; intuitively, it indicates that the sub-circuit is OK at this point even if all stars (zero or more repetitions) became plusses (one or more repetitions). Inductively, given an extended regular expression:

Base Case: If the expression is a primitive expression, build the combinational logic to evaluate the Boolean formula for the primitive expression. Let f denote the output of this formula. The enable output e is equal to the activate input a_i. Both ok and ok_p are set to $a_i \wedge f$. The activate-out signal is $a_i \wedge f$ delayed by one clock signal (one flip-flop in the circuit).

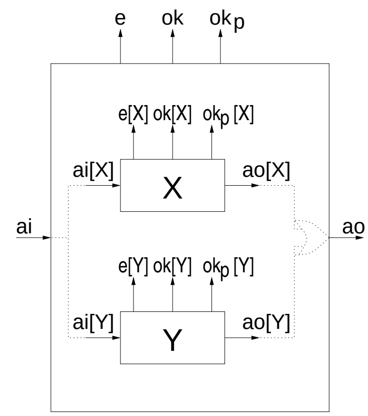


Figure 4: Circuits are built recursively from the circuits for their sub-expressions. The dotted lines show the construction for the activation signals for the choice operator $X||Y$.

Choice: If X and Y are regular expressions with corresponding circuit translations, then build the circuit for X `||` Y from the circuits for X and Y as follows: (Denote the signals for X's circuit with $[X]$, similar for Y. See Figure 4.)

$$a_i[X] = a_i \qquad a_i[Y] = a_i \qquad a_o = a_o[X] \vee a_o[Y]$$

connects that activation signals, and:

$$e = e[X] \vee e[Y] \qquad ok = ok[X] \vee ok[Y] \qquad ok_p = ok_p[X] \vee ok_p[Y]$$

generates the enable and OK signals.

Sequence: Similarly, build the circuit for X `,` Y as follows:

$$a_i[X] = a_i \qquad a_i[Y] = a_o[X] \qquad a_o = a_o[Y]$$

connect the activation signals so that X goes first, and then activates Y in sequence. The constructions $e = e[X] \vee e[Y]$ and $ok_p = ok_p[X] \vee ok_p[Y]$ are intuitive — the circuit is enabled or "OK-plus" if either sub-circuit is enabled or OK-plus. The OK signal needs extra clauses

$$ok \;=\; (e[X] \Rightarrow ok[X]) \wedge (e[Y] \Rightarrow ok[Y]) \wedge (ok[X] \vee ok[Y])$$

because X or Y might consist of a Kleene star, and the construction for the Kleene star is always OK as soon as the circuit is activated, regardless of the values on the interface wires (because the star allows matching zero copies of the repeating expression). The extra clauses prevent these empty-match OK signals from propagating erroneously.

Pipeline: For X `@` Y, all of X's signals are connected to the corresponding signals for the circuit for X `@` Y, since the current thread ignores Y. At the same time, a new thread for Y gets activated: $a_i[Y] = a_o[X]$.

Kleene Star: For X^*, connect the activation signals as $a_i[X] = a_i \vee a_o[X]$ and $a_o = !ok_p[X] \wedge (a_i \vee a_o[X])$. The enable output is $e = e[X] \vee a_o[X]$, the OK output is $ok = ok[X] \vee a_i \vee a_o[X]$, and the OK-plus output is $ok_p = ok_p[X]$. Because of the repetition, the circuit self-activates, so the $a_o[X]$ signal appears in several formulas. The Kleene star accepts the empty string, so the a_i signal appears combinationally in the equations for ok and a_o (as well as indirectly in e for the first cycle of X's activation). Here, we see the use of the ok_p signal: the activate output is disabled if X is truly matching the interface (rather than vacuously matching because of a Kleene star). The deterministic choice restriction prevents the case where a_o should be true at the same time as $ok_p[X]$.

Earlier, we imposed three restrictions to allow efficiently building a monitor: (1) normalization to eliminate empty strings within Kleene stars, (2) deterministic choice, and (3) one thread per pipeline stage. The first two can be handled statically using standard techniques. To enforce the third restriction, we augment our monitor to generate a pipeline-violation error. Intuitively, for each pipeline operator X @ Y, trying to activate Y when it is already running generates a pipeline-violation error. A complication, however, is that the signals from the already running thread and the new activation can interfere. The easiest way around this complication is to generate three versions of every signal in the construction above: the regular version as described already; a primed version, which ignores any new activation; and a double-primed version, which tracks only the first cycle of a new activation. The formulas for the primed signals are identical to the ones above, with primed signal names replacing unprimed signal names, except for the initial thread activate signal $a_i'[Y] = \text{false}$ instead of $a_i[Y] = a_o[X]$, and at the base case latches a_o' is driven by the same flip-flop (unprimed) that drives a_o. The formulas for the double-primed signals are also identical to the regular signals, except with double-primed signal names and the base case a_o'' is always false rather than driven by the flip-flop. (Considerable redundancy could be eliminated, but this construction is easy to explain and implement.) A pipeline-violation error occurs whenever $a_i[Y] \wedge ok_p'[Y]$. The thread enable and ok signals are defined as $tenable = e[Y]$ and $tok = (e'[Y] \Rightarrow ok'[Y]) \wedge (e''[Y] \Rightarrow ok''[Y])$.

The size of the generated circuit and the runtime of the algorithm are both linear in the size of the (macro-expanded) specification, because the construction does constant work and generates constant-sized circuitry for each operator in the specification. The top-level $tenable_i \Rightarrow tok_i$ circuitry can be accounted for in the cost of the pipeline operators.

We have implemented the above translation into a tool that outputs Verilog or VHDL. Generating any output format that is as expressive as sequential circuits should be straightforward.

5. CONCLUSION AND FUTURE WORK

We have presented a novel, high-level specification style for interface monitors, as well as a linear-size, linear-time translation algorithm into monitor circuits. The specification style naturally fits the interfaces used between IP blocks in systems-on-chip. We hope that these results will facilitate the use and broaden the adoption of monitor-based verification methodologies.

In the short term, we need to improve the translation tool. Our current implementation is rather crude. A revised, more robust tool would be better suited for public distribution. In addition, many optimizations are possible and should be implemented. To access the tool flows of other verification researchers, our tool will have to be able to translate specifications into the lower-level specifications used by other tools. In particular, we do not anticipate difficulties translating from our specifications into the specifications used in the interface-monitor research that inspired this work [8, 13, 7].

The macro-expansion of productions can blow-up the size of the regular expressions. Additional research is needed to see if there are practical ways to avoid this problem, such as by introducing new language features to simplify the expressions or better translations that schedule reuse of circuitry.

On the theoretical side, the precise semantic model and expressive power of our specification style are worth exploring. Is our unrestricted specification style exactly equivalent to alternating automata? How much power do we lose by imposing restrictions? Are there other trade-offs between expressive power and efficiency that would be practically useful?

Exploring the theory and semantics would also clarify obscure points in our work. For example, in our current language, the interaction between storage variables and pipelining is analogous to multiprogramming with shared variables. If a variable is used by only a single thread, the results are intuitive, but if multiple threads access a variable, subtle interactions can result. The language may need to be augmented with different classes of storage variables that behave differently with respect to pipelining. In some cases, we want the current behavior, with threads able to communicate across pipeline stages by accessing shared variables. In other cases, we want each pipeline stage to have its own copy of a variable, with values forwarded from one stage to the next when threads are activated. In this model, the variable holds values that follow a transaction through the pipeline.

An important line of future work is to gain experience with this specification style on additional, real interface protocols. Applying the specification style in practice is the only way to validate the adequacy of the specification style, or determine what additional features are needed.

6. REFERENCES

[1] ARM Limited. *AMBA Specification (Rev 2.0)*. 13 May 1999.

[2] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. *Tools and Algorithms for the Construction and Analysis of Systems: 8th Intl Conf*, pp. 296–311. LNCS 2280. Springer, 2002.

[3] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, Y. Rodeh. The temporal logic sugar. *Computer-Aided Verification: 13th Intl Conf*, pp. 363–367. LNCS 2102. Springer, 2001.

[4] L. Bening, H. Foster. *Principles of Verifiable RTL Design*. 2nd Ed. Kluwer, 2001.

[5] A. K. Chandra, D. C. Kozen, L. J. Stockmeyer. Alternation. *JACM*, 28(1):114–133, 1981.

[6] E. M. Clarke, E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Workshop on Logics of Programs*, pp. 52–71, 1981. Published as LNCS 131. Springer, 1982.

[7] M. S. Jahanpour, E. Cerny. Compositional verification of an ATM switch module using interface recognizer/suppliers (IRS). *Intl High-Level Design, Validation, and Test Workshop*, pp. 71–76. IEEE, 2000.

[8] M. Kaufmann, A. Martin, C. Pixley. Design constraints in symbolic model checking. *Computer-Aided Verification: 10th Intl Conf*, pp. 477–487. LNCS 1427. Springer, 1998.

[9] O. Kupferman, M. Y. Vardi. Module checking. *Computer-Aided Verification: 8th Intl Conf*, pp. 75–86. LNCS 1102. Springer, 1996.

[10] P. Raymond. Recognizing regular expressions by means of dataflow networks. *23rd Intl Coll on Automata, Languages, and Programming*, pp. 336–347. LNCS 1099. Springer, 1996.

[11] A. Seawright, F. Brewer. High-level symbolic construction techniques for high performance sequential synthesis. *30th Design Automation Conf*, pp. 424–428. ACM/IEEE, 1993.

[12] K. Shimizu, D. L. Dill, C.-T. Chou. A specification methodology by a collection of compact properties as applied to the Intel Itanium Processor Bus protocol. *Correct Hardware Design and Verification Methods: 11th IFIP WG 10.5 Adv Research Working Conf*, pp. 340–354. LNCS 2144. Springer, 2001.

[13] K. Shimizu, D. L. Dill, A. J. Hu. Monitor-based formal specification of PCI. *Formal Methods in Computer-Aided Design*, pp. 335–353. LNCS 1954. Springer, 2000.

[14] Sonics Incorporated. *Open Core Protocol Specification 1.0*. Document Version 1.2.

[15] J. Yuan, K. Shultz, C. Pixley, H. Miller, A. Aziz. Modeling design constraints and biasing in simulation using BDDs. *Intl Conf on Computer-Aided Design*, pp. 584–589. IEEE/ACM, 1999.