A Design and Validation System for Asynchronous Circuits

Peter Vanbekbergen, Albert Wang and Kurt Keutzer
Synopsys, Inc.
Mountain View, CA, 94043

Abstract—

In this paper we present a complete methodology for the design and validation of asynchronous circuits starting from a formal specification model that roughly corresponds to a timing diagram. The methodology is presented in such a way that it is easy to embed in the current methodology for synchronous circuits. The different steps of the synthesis process will just be briefly touched upon. The main part of the paper concentrates on the simulation and validation of asynchronous circuits. It discusses where the designer needs validation and how it can be done. It also explains how this process can be automated and embedded in the complete methodology.

I. Introduction

Design technology to support the system-level design of application-specific microelectronics systems is rapidly becoming an important area. Today, a designer has an extensive set of tools at his disposal for the design of synchronous circuits, where a single clock controls and synchronizes all collective activity. These tools do not only support synthesis of these circuits at different abstraction levels but also support the verification and simulation. These tools fit together to offer the designer a complete methodology for the design and verification of synchronous designs.

As we rapidly move to system-level solutions, it becomes clear that the paradigm of one central clock ruling the activity of the complete system is not valid any more. The system becomes partitioned into different synchronous islands, each with their own clock. While the synchronous paradigm still applies to these "islands", it does not apply any more to the circuits that control the communication between the different "islands". These interface circuits are inherently "asynchronous", because they do not use a clock, or because not all signals in the interface follow the same clocking methodology.

Even when the use of asynchronous circuits is not necessary, they offer particular advantages, important in many applications. Asynchronous circuits can have a much lower latency than their synchronous counterparts, because they can respond to an input change without waiting for the clock. This is a desirable property in many interface applications. Asynchronous circuits also play an important role for low power circuits. Systems are being designed today, completely based on the asynchronous paradigm [10], although that mixed-mode solutions, where only small part of the chip is asynchronous, are currently more common [3] in industrial designs.

The lack of a complete asynchronous design methodology not only represents a serious problem when an asynchronous design cannot be avoided, but also has as a consequence that designers are reluctant to design asynchronous circuits which often results in suboptimal solutions. Due to the lack of a formal specification method for asynchronous circuits, designers often make hidden assumptions, that are not indicated in the specification. This makes it hard to use consistent synthesis and verification methods. Moreover communicating specifications between designers becomes tricky as each designer tends to have his own assumptions. Although the latest research addresses a lot of isolated problems in the design of asynchronous circuits, no complete methodology has been presented. It is exactly this gap that this paper tries to bridge.

In this paper we present a complete methodology for the design and validation of asynchronous circuits starting from a formal specification model that roughly corresponds to a timing diagram. The methodology is presented in such a way that it is easy to embed in the current methodology for synchronous circuits. The different steps of the synthesis process will just be briefly touched upon. The main part of the paper concentrates on the simulation and validation of asynchronous circuits. It discusses where the designer needs validation and how it can be done. It also explains how this process can be automated and embedded in the complete methodology.

II. Methodology

The following aspects are key to embedding a new methodology for the design of asynchronous circuits in the current methodologies.

- **Specification:** A formal specification methodology is essential for any design methodology and it may not be obscure to designers. Therefore it is necessary to support different specification methodologies, including the ones designers are more familiar with, even if these methodologies are not the most general ones.
- **Simulation & validation:** Simulation plays a central role in the design of any circuit and asynchronous circuits are no exception. Even if a formal verification tool is available the methodology still needs to support simulation.
- **Synthesis:** The synthesis procedure needs to guarantee circuit correctness. Correctness also includes analyzing and verifiying the exact conditions under which the circuit will work. For instance, in a synchronous environment this corresponds to determining the maximum clock speed that is allowed. In an asynchronous circuit this corresponds to determining when a new input may be applied to the circuit. Having implicit conditions under which the circuit works is unacceptable.
  The model for synthesis and verification of the asynchronous circuits needs to be consistent with the model used for synchronous circuits. The delay-insensitive model is very popular for asynchronous designs [5]. It assumes no delays in the wires and unbounded delays in the gates. This is not consistent with the synchronous model that assumes bounded delays in the wires and gates.
- **Library:** The methodology needs to support standard libraries. The synthesis procedures may not solely rely on special elements (like c-elements or specially designed flip-flops) that are not supported in standard libraries.

32nd ACM/IEEE Design Automation Conference ®

An overview of the proposed methodology is presented in Figure 1. The shaded boxes represent the information that has to be provided by the designer. First of all there is the specification itself which may be a Signal Transition Graph (STG) [2] or an asynchronous finite state machine model (AFSM) [6], [11]. The compiler translates an AFSM into an STG. The STG-model is informally introduced in Section III. Second, there is the library of gates the circuit is going to be mapped to. This library may just be a standard library. The designer must ensure that the cells that are used in the mapping process are hazard-free. The classical and- and or-gates are usually hazard-free, but many other cells, like multiplexers, may contain internal hazards. The designer must disable such cells for use during the mapping process. Finally, the designer must provide a tolerance factor that indicates how much the delay of a gate may vary around the normal value. In this way the designer can take into account variations of the process parameters. He can also take into account delay variations that are introduced by routing in the layout. It is obvious that low tolerance factors result in efficient but less robust circuits, while high tolerance factors result in less efficient but more robust circuits. The initial STG specification, library information and tolerance factor are the only data that have to be provided by the designer. All other data are generated automatically by the compiler.

The finite state machine model is a clear, easy-to-analyze model that designers are familiar with. The signal transition graph model however is more general in handling concurrency, conditional behavior and timing. Although designers are used to timing diagrams, which the STG-model is close to, they are not willing to immediately adopt this new model. Thus both models are supported in this compiler. Actually any specification model that can be translated into a state graph can be handled by the compiler because all the synthesis algorithms work at state graph level (only the simulation-environment needs the STG-model).

Another advantage of these formal specification models is that they also specify the allowed behavior of the environment. It gives the synthesis algorithms the opportunity to come up with better implementations and gives the simulation environment the opportunity to verify whether circuit *and environment* are behaving as specified.

From the initial STG VHDL-code is generated that can be simulated in different ways. Simulations validate that the initial specification corresponds to what the designer has in mind and that the environment of the asynchronous circuit is correctly modeled in the STG. This will be discussed in Section IV.

The STG is transformed into a state graph internally in the compiler. The state graph represents all the possible states of the circuit It is therefore much more elaborate than the STG-model or FSM-model.

This state graph may still contain state assignment conflicts. So state signals are added to the state graph to eliminate all state assignment conflicts, resulting in a new state graph. This is an adaptation of the technique proposed in [9].

Once a state graph is obtained logic equations may be derived, based on the technique presented in [4].

Based on the information in the state graph and in the technology independent equations, all potential hazards can be identified. This includes static-0 static-1 and dynamic hazards [4]. The technique for the analysis of static hazards and dynamic hazards is an adaptation from [4]. Our simulations have shown that dynamic hazards do play an important role.

The technology independent equations are then mapped to the library. Based on the timing information in the library and the tolerance factor, a timing analysis step determines the exact timing relations that

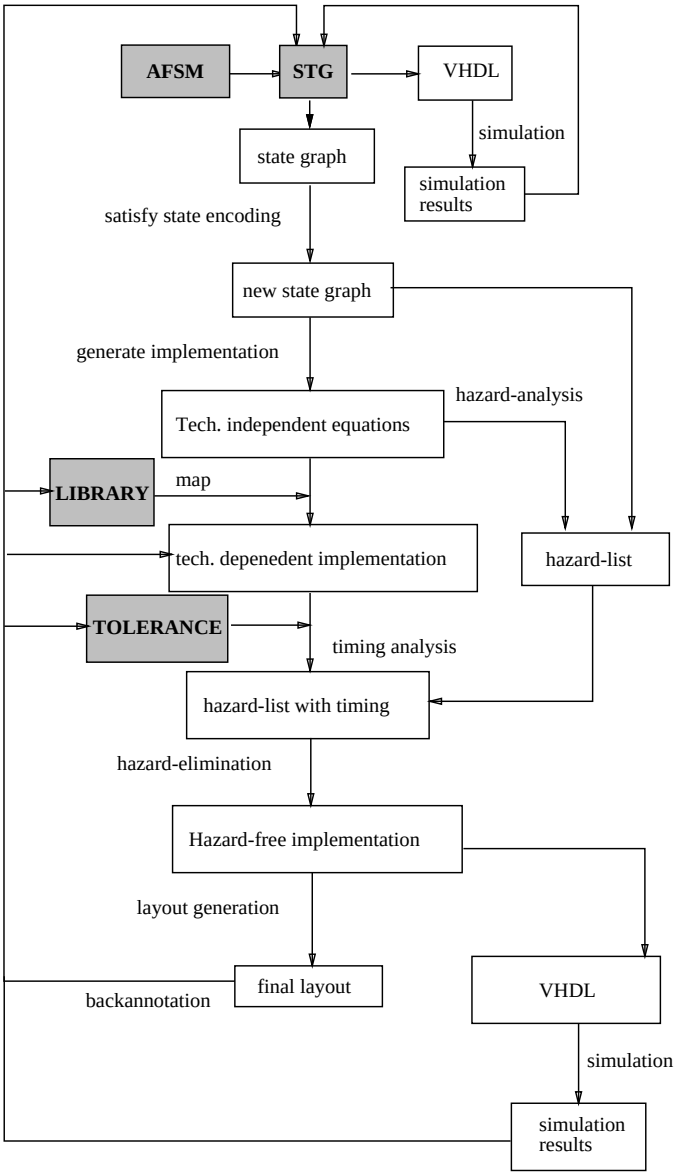


Fig. 1. An overview of the methodology.

eliminate all the hazards in the hazard-list. The technique will not only delay the circuit by adding delay-elements, but it will also produce the exact timing relations for the environment (input signals) for the circuit to operate correctly. In a synchronous circuit this would correspond to determining the maximum clock speed.

Finally the layout is generated. The layout information can be backannotated on the circuit. Timing analysis can then verify if all hazards remain eliminated. If not, new delays may be added by the hazard-elimination procedure and a new layout may be generated. If this process fails to converge or converges slowly, the designer can also increase the tolerance factor to account for the routing delays. The complete circuit can then be simulated in different ways as discussed in Section IV.

III. The Specification Model

An STG corresponds to a Petri net [8], [2]. A Petri net is a directed graph with two different types of vertices: places (open circles) and

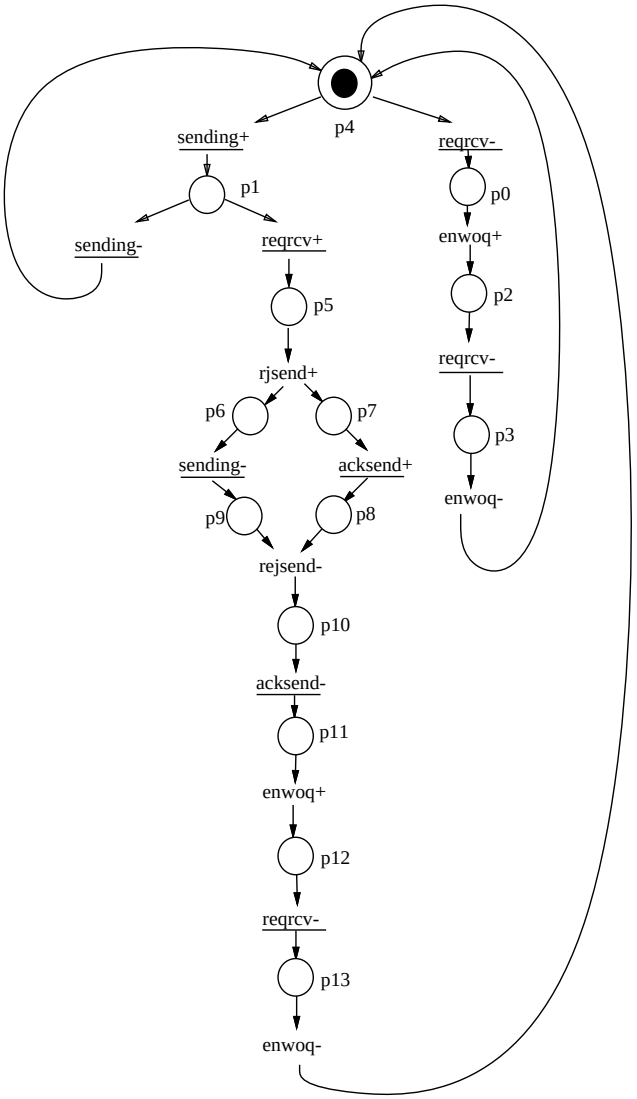


Fig. 2. The STG for rcv-setup.

transitions. Places determine the (causal) relationships between the transitions. Transitions correspond to the signal transitions of the input and output signals of the circuit. Transitions of input signals are underlined.

Execution of a Petri net is controlled by the position and movement of markers (called *tokens*) in the Petri net. Tokens, indicated by black dots, reside in the circles representing the places of the net. Tokens are moved by the firing of the transitions of the net. A transition is enabled when all its input places carry a token. An enabled transition may fire. This is done by removing the enabling tokens from their input places and generating new tokens which are deposited in the output places of the transition. In Figure 2, $sending^+$ is enabled since it has a token in its input place p_4. If $sending^+$ fires, the token in p_4 will be removed and a new token will be placed in place p_1. The distribution of tokens in a Petri net defines the state of the net at each moment in time and is called its *marking*.

The STG can model conditional behavior as well as concurrent behavior. A choice is modeled by place p_4. In this case $sending^+$ can fire or $reqrcv^+$ can fire, but not both. They determine a choice

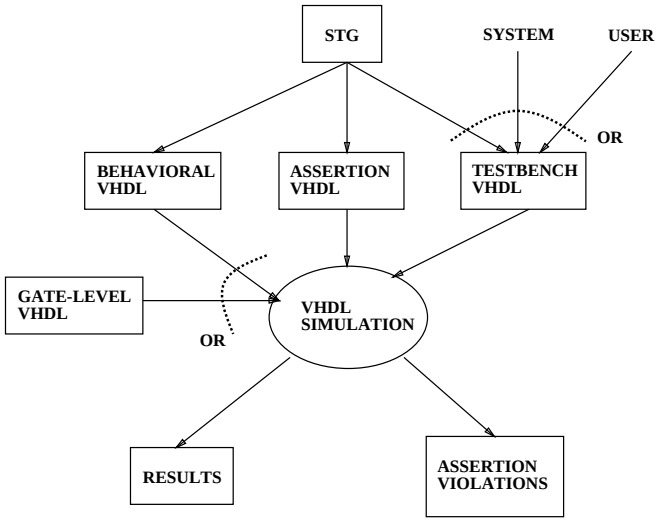


Fig. 3. An overview of the simulation environment.

made by the environment. The transitions $sending^-$ and $acksend^+$ can fire concurrently after $rjsend^+$ has been fired.

The result of the synthesis process is shown in Figure 6. Note that delay lines have been added to slow down parts of the circuit to eliminate hazards. Not that it is also indicated how much the input signals (and thus the environment) should be slowed for the circuit to operate correctly.

IV. Simulation

Simulation serves two purposes. First, simulating the behavioral specification is necessary to convince the designer that his specification corresponds to what he/she has in mind. But in this case, the designer also needs to validate that the assumptions he made about the environment in which this circuit is going to be embedded correspond to the real environment.

Second, the simulation at the gate-level after synthesis convinces the designer that his own implementation or synthesized implementation is correct.

With all this in mind a designer needs to juggle around with four different types of VHDL (Figure 3). **Behavioral VHDL** that models the behavior of the specification. **Testbench VHDL** that generates the input vectors for simulation. **Assertion VHDL** that checks if the circuit and environment are responding as specified. An finally, **Gate-level VHDL** derived from the designed circuit. Usually the gate-level VHDL is the only VHDL available to the designer. Due to a lack of a consistent specification model writing down the other VHDL types is impossible or too time-consuming. In this methodogy all these VHDL-files can be generated directly from the STG and synthesis results, without designer intervention. This means that before and after synthesis simulation can occur automatically and any violations of the initial specification during the simulation are immediately reported to the designer.

The architecture is shown in Figure 4. The behavioral VHDL takes its inputs from the testbench and produces the outputs. The testbench VHDL produces the inputs for the behavioral VHDL, but it also has as its inputs the outputs of the behavioral VHDL. Note that also the environment can be reactive, waiting for the asynchronous circuit to respond. The assertion VHDL has as its inputs the inputs as well as the outputs of the behavioral VHDL. The assertion VHDL monitors the behavior of these signals and produces error messages if the initial

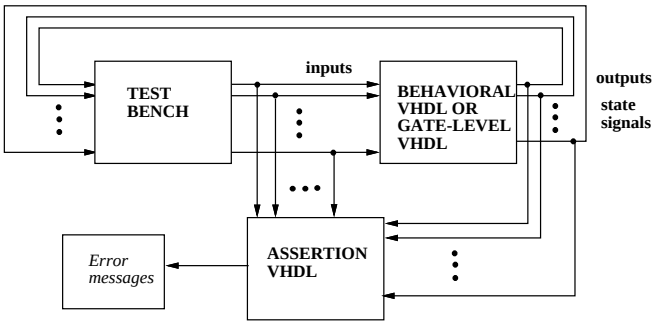


Fig. 4. The simulation architecture.

specification is violated.

It might be interesting to note that simulation is also an important tool for debugging the compiler. Many software and theoretical bugs were discovered this way during development of the compiler.

For the coming subsections describing the different VHDL-styles, we will refer to the STG presented in Figure 2.

A. Behavioral VHDL.

From the STG VHDL-code can be generated which represents the behavior of the STG. The main characteristic of an STG (or Petri net) is that each place can be considered a state. This corresponds to a machine with more than one active state [7].

```
rejsend_down: BLOCK ((places(8) = '1' AND
                      places(9) = '1'))
BEGIN
  rejsend_int <= GUARDED '0';
  places(8) <= GUARDED '0' ;
  places(9) <= GUARDED '0' ;
  places(10) <= GUARDED '1' ;
END block rejsend_down;
```

The dynamic behavior of the places is represented by a vector of booleans. If a token arrives in place p (with p an integer) then places(p) is set to one. If a token is removed from the place places(p) is set to zero. This can easily be extended for multiple tokens.

Because transitions can potentially be concurrent with any other transition in the STG, each transition is modeled by a different (guarded) block. The guard-expression corresponds to the enabling condition of the transition. It indicates that the transition can fire if places 8 and 9 contain a token.

The transition firing and the token-flow are modeled inside the block. It indicates that $rejsend$ is set to zero, that the tokens in places 8 and 9 are removed and that a token is placed in place 10.

```
sending_down_3: BLOCK ((places(1) = '1' AND
                        sending = '0'))
BEGIN
  places(1) <= GUARDED '0' ;
  places(4) <= GUARDED '1' ;
END block sending_down_3;
```

The model is somewhat different for input signals. In this case we are not generating a signal transition, but waiting for a signal transition to take place. So in this case, it is indicated in the guard that not only place 1 needs to contain a token, but the input-signal $sending$ needs to go low before we can execute the block command.

In these specifications there is more than one guarded assignment to the same variable. That is why a resolution function is needed here. An example is shown below. Although that there are multiple assignments, the guard expressions should make sure that this never happens. If more than one assignment is active simultaneously, it means a high and low transition is being enforced on the same signal at the same time. This corresponds to an inconsistent initial specification. An assert statement was added to check for this.

```
TYPE std_vector IS
  ARRAY (NATURAL RANGE <>) OF std_logic;
FUNCTION oring_std ( drivers : std_vector)
  RETURN std_logic;
SUBTYPE ored_std IS
  oring_std std_logic;
TYPE ored_std_vector IS
  ARRAY (NATURAL RANGE <>) OF ored_std;

  FUNCTION oring_std ( drivers : std_vector)
    RETURN std_logic IS
    VARIABLE accumulate : std_logic := '0';
    BEGIN
      FOR i IN drivers'RANGE LOOP
        accumulate := accumulate OR drivers(i);
      ASSERT (i < 2);
      END LOOP;
    RETURN accumulate;
  END oring_std;
```

B. Testbench VHDL.

For the testbench VHDL there are many alternatives. The user may decide to write it himself, to test the circuit for situations he considers to be critical. The compiler could generate special testbenches to activate all possible hazards, for example.

In our current implementation we have chosen to test the circuit in such a way that new transitions are generated from the moment that they are allowed to fire according to the specification. This simulation tries to operate the circuit at the maximal possible speed.

This can be accomplished by switching input and output signals and generating the behavioral VHDL code the same way as in the previous section. However in this case our assumption that a place that has multiple output transitions, only has output transitions of input signals is not valid any more. This leads to VHDL-code that does not behave according to the STG specification.

```
reqrcv_up_0: BLOCK ((places(4) = '1'))
BEGIN
  places(4) <= GUARDED '0' ;
  places(0) <= GUARDED '1' ;
END block reqrcv_up_0;
sending_up_2: BLOCK ((places(4) = '1'))
BEGIN
  places(4) <= GUARDED '0' ;
  places(1) <= GUARDED '1' ;
END block sending_up_2;
```

Both $sending^+$ and $reqrcv^+$ are enabled but only one of them is supposed to fire. In the above VHDL code, both will fire at the same time. Here we need an additional mechanism to control conditional behavior.

```
reqrcv_up_0: BLOCK (places(4) = '1' AND
                    place_0_int = 0)
BEGIN
```

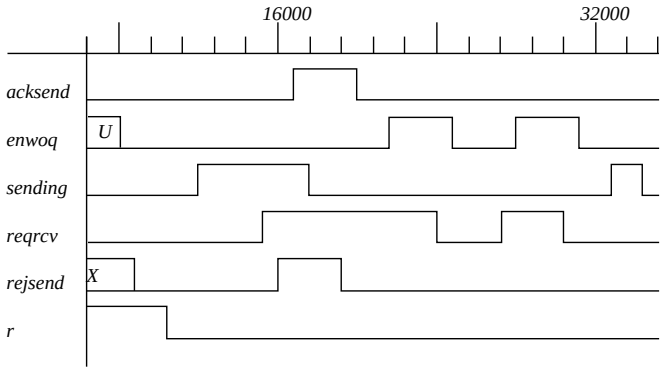


Fig. 5. The simulation of rcv-setup.

```
  place_0_int <= GUARDED 1 ;
  reqrcv_int <= GUARDED '1' ;
  places(4) <= GUARDED '0' ;
  places(0) <= GUARDED '1' ;
END block reqrcv_up_0;

sending_up_2: BLOCK (places(4) = '1' AND
                     place_0_int = 1)
BEGIN
  place_0_int <= GUARDED 0 ;
  sending_int <= GUARDED '1' ;
  places(4) <= GUARDED '0' ;
  places(1) <= GUARDED '1' ;
END block sending_up_2;
```

For each place that has more than one output transition an additional variable is declared ($place\_0\_int$). This variable selects only one of the output transitions. After one of the transitions has been fired this variable is set such that is selects the other transition. In this way none of the transitions is favored. However more complex schemes may be required to guarantee that every conditional path in the STG is activated. This is the subject of current research.

In many cases it is necessary to slow down the environment to make sure that the circuit operates correctly. This information is the output of the hazard-elimination process (Figure 6). So in many cases some of the signals have be slowed down using the "AFTER" statement. This is only necessary for those testbenches that are going to be used for testing the final circuit implementation.

```
rejsend_down: BLOCK ((places(8) = '1' AND
                      places(9) = '1'))
BEGIN
  rejsend_int <= GUARDED '0' AFTER 3100 PS;
  places(8) <= GUARDED '0' AFTER 3100 PS;
  places(9) <= GUARDED '0' AFTER 3100 PS;
  places(10) <= GUARDED '1' AFTER 3100 PS;
END block rejsend_down;
```

C. Gate-level VHDL.

After the synthesis has been done and a gate-level desription has been generated, this can be translated into gate-level VHDL. This can be plugged in the architecture presented in Figure 4 instead of the behavioral VHDL-code.

D. Assertion VHDL.

A block is also needed that checks if everything that occurred during simulation does not violate the initial specification. This means not only checking if the outputs of the circuit behave according to the specification but also the input signals, generated by the testbench. This is important if the testbench contains user-defined vectors or when the testbench corresponds to the complete system in which the asynchronous circuits will plug into. In this way the designer can not only find out if something is going wrong but can also determine what the cause of the problem is. He can immediately identify whether there is an error in the asynchronous circuit or whether the environment is not behaving as specified in the STG.

```
ASSERT (enwoq'STABLE=TRUE  OR
        (places(0) = '1' AND enwoq = '1') OR
        (places(3) = '1' AND enwoq = '0') OR
        (places(11) = '1' AND enwoq = '1') OR
        (places(13) = '1' AND enwoq = '0'))
REPORT
"output enwoq not according to spec"
SEVERITY NOTE;
```

This assertion module also contains all the statements that describe the token-flow as indicated in section 5. But it also contains assertion statements that indicate that the signal is supposed to be stable except if one of its transitions is enabled, and it makes the right transition. So if place 0 contains a token $enwoq$ is allowed to go high.

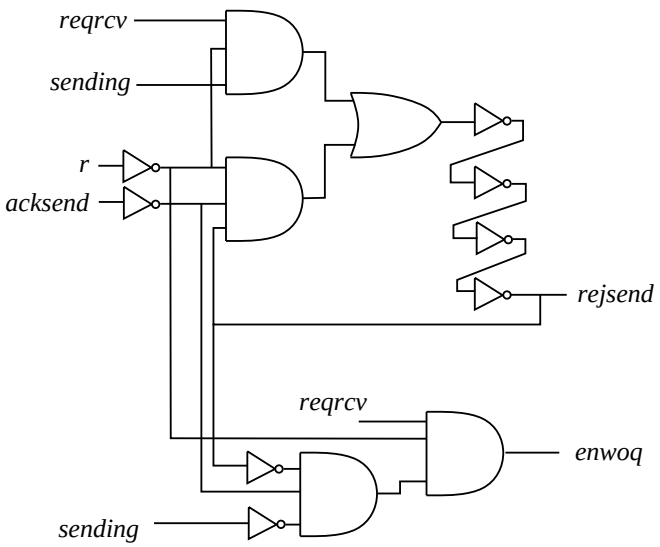
E. The speed of an asynchronous circuit.

Until now no successful measure of the speed of an asynchronous circuit was given. Measures presented in for instance [1], [9] were only valid for marked graphs. Therefore we propose the following practical measure. Just count the transitions that the complete system makes during simulation for a fixed time interval. This is a valid measure, because the architecture as set up in Figure 4 and way the testbench is set up make sure that new input transitions are generated as fast as possible.

F. Experimental Results.

The experimental results are presented in Table I. The second column represents the number of lines of STG code. The third column contains the number of lines of VHDL code, only containing the testbench and assertion module. The third row represents the number of signal transitions for a simulation of 200ns if the tolerance factor is set to 0.2. This means that the minimum delay of a gate corresponds to 80% of its mean value, while the maximum delay of gate becomes 120% of its mean value. The fourth row represents the number of toggles for a simulation of 200ns if the tolerance factor is set to 0.5. The following conclusions can be drawn from this experiment.

- Although the VHDL-code is necessary for simulation, it becomes painful to write down for a designer. So it pays to automate the process of deriving VHDL code from the initial STG specification. The size of the VHDL-code is about a factor 5 bigger than the STG-code.
- The simulation can be used as an overall measure for the speed. It becomes clear that by setting the tolerance factor high, the circuit becomes more robust. However the overall speed of the circuit goes down drastically.



reqrcv 3.1ns, acksend 1.06ns, sending 2.3ns

Fig. 6. The synthesized circuit rcv-setup.

Spec.	STG	VHDL	t(0.2)	t(0.5)
alloc-outbound	22	312	176	132
atod	21	153	192	147
dff	60	207	107	48
ebergen	21	145	104	50
fifo	15	103	134	84
full	15	113	212	93
isend	79	387	122	84
it-control	30	446	127	89
mp-fwd-pkt	38	161	195	147
nak-pa	47	177	215	173
nousc-ser	13	89	215	129
nowick	23	163	235	184
pe-rcv-ifc	110	391	124	86
pe-rcv-ifc-prs	117	413	134	86
pe-send-ifc	116	406	117	100
qr42	21	161	102	49
ram-read-sbuf	51	195	254	192
rcv-setup	48	149	154	120
rf-control	30	317	127	107
rlm	23	142	352	265
rpdft	29	187	146	123
sbuf-ram-write	48	201	193	167
sbuf-read-ctl	42	576	212	158
sbuf-send-ctl	60	216	214	147
sbuf-send-pkt2	68	237	86	58
sendr-done	25	87	223	119
trimos-send	25	199	116	54
vbe10b	29	223	166	102
vbe4a	19	137	248	194

TABLE I
EXPERIMENTAL RESULTS.

V. CONCLUSIONS

In this paper we have introduced a complete methodology for the design and validation of asynchronous circuits. The methodology is easy to embed in current design methodologies for synchronous circuits.

The emphasis in this paper was on the validation of the initial specification and the final circuit. It is shown how all the VHDL-code, necessary for the automatic simulation and validation, can be derived from the STG. It was also emphasized that this code not only checks the behavior of the final circuit, but also checks if the environment behaves as specified in the initial STG. This is crucial when the circuit is plugged into its final environment. Finally, it is argued that this validation environment can be used for general speed measurements of the asynchronous circuit.

A lot of research remains to be done. The VHDL code needs to be extended so that more complex timing and conditional constructs can be taken into account. The VHDL code for the testbench needs to be extended in such a way that all possible hazards can be activated during simulation.

VI. ACKNOWLEDGEMENTS

The authors wish to thank Prof. Lavagno for his insights and many stimulating discussions.

REFERENCES

[1] S. M. Burns. Sizing asynchronous circuits produced by martin synthesis. In *ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, August 1990.

[2] T. A. Chu. *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, MIT, June 1987.

[3] Carl Larson. Making desktop computers energy stars. In *EDN Products Edition*, November 1993.

[4] Luciano Lavagno. *"Algorithms for Synthesis and Testing of Asynchronous Circuits"*. Kluwer Academic Publisher, 1993.

[5] A. Martin. Synthesis of asynchronous VLSI circuits. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North-Holland, 1990.

[6] S. M. Nowick and D. L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proceedings of the International Conference on Computer-Aided Design*, pages 318–321, November 1991.

[7] Douglas E. Perry. *VHDL*. McGraw-Hill, 1991.

[8] James L. Peterson. *"Petri Net Theory and the Modeling of Systems"*. Prentice-Hall, 1981.

[9] P. Vanbekbergen. *Synthesis of Asynchronous Control Circuits from Graph-Theoretic specifications*. PhD thesis, Catholic University of Leuven, ESAT, September 1993.

[10] K. Vanberkel. *Handshake Circuits: an Intermediary between communicating processes and VLSI*. PhD thesis, Technical University Eindhoven, 1992.

[11] K.Y. Yun and D. L. Dill. Unifying synchronous/asynchronous state machine synthesis. In *Proceedings of the International Workshop on Logic Synthesis*, May 1993.