Asynchronous Interface Specification, Analysis and Synthesis*

Michael Kishinevsky
Intel Corporation
Hillsboro, OR, USA

Jordi Cortadella
Technical University of Catalonia
Barcelona, Spain

Alex Kondratyev
The University of Aizu
Aizu-Wakamatsu, Japan

Abstract

Interfaces, by nature, are often asynchronous since they serve for connecting multiple distributed modules/agents without common clock. However, the most recent developments in the theory of asynchronous design in the areas of specifications, models, analysis, verification, synthesis, technology mapping, timing optimization and performance analysis are not widely known and rarely accepted by industry.

The goal of this tutorial is to fill this gap and to present an overview of one popular systematic design methodology for design of asynchronous interface controllers. This methodology is based on using Petri nets (PN) *a formal model that, from the engineering standpoint, is a formalization of timing diagrams (waveforms) and from the system designer standpoint is a* concurrent *state machine, in which local components can perform independent or interdependent concurrent actions, changing their local states asynchronously. We will introduce this model informally based on a simple example: a VME-bus controller serving reads from a device to a bus and writes from the bus into the device.*

1 Specification with Petri Nets

1.1 From timing diagrams to Petri Nets

Figure 1 depicts the interface of a device with a VME bus. The behavior of the controller is as follows: a request to read from or write in to the device is received by one of the signals DSr or DSw respectively. In a read cycle, a request to read is done through signal LDS. When the device has the data ready ($LDTACK$), the controller must open the transceiver to transfer data to the bus (signal D). In the write cycle, data is first transferred to the device. Next, a request to write is done (LDS). Once the device acknowledges the reception of the data ($LDTACK$) the transceiver must be closed to isolate the device from the bus. Each transaction must be completed by a return-to-zero of all interface signals, seeking for a maximum parallelism between the bus and the device operations. Figure 2 shows a timing diagram of the

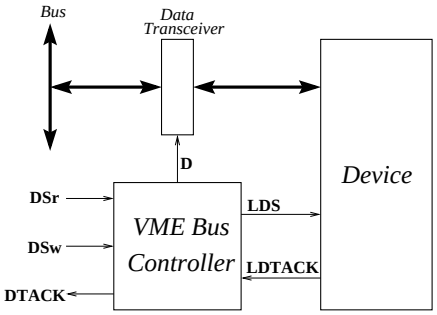


Figure 1: VME bus controller

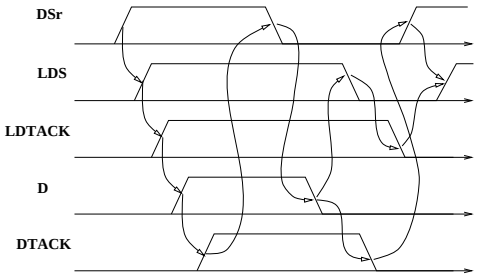


Figure 2: Waveforms for the READ cycle

read cycle and Figure 3 the corresponding Petri Net. All events in this Petri Net are interpreted as signal transitions: rising and falling signal transitions are labeled with "+" and "−" respectively. Petri Nets with such signal interpretations are called *Signal Transition Graphs (or* STGs*)* [17].

A PN has two types of vertices: places (denoted by circles) and transitions (denoted by boxes), and arcs from places to transitions and from transitions to places. Places correspond to local states of the system and are used for keeping information about system resources and conditions for execution of transitions. Places can keep tokens (denoted by black dots). A token in a place indicates that a resource is available or a condition satisfied. In general more than one token can be kept in a place, but we will consider only the simplest case: a place cannot contain more than one token. A set of all places currently marked with a token corresponds to a current *global* state of the net. Such global states are called *markings*. The initial marking of the PN in Figure 3 is $\{p_0, p_1\}$.

1.2 Token game and concurrency

Transitions correspond to system events (signal transitions in the example). A transition is *enabled* if all input places contain a token. In the initial marking of the PN in Figure 3

*Work partially supported by ACiD-WG (Esprit 21949) and CICYT TIC 95-0419

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

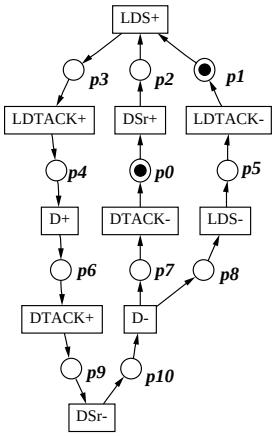


Figure 3: STG for the READ cycle

only one transition, $DSr+$, is enabled; another one, $LDS+$, is not: only place p_1 among two of its input places, p_1 and p_2, contains a token. Every enabled transition can fire. Firing removes one token from every input place of the transition and puts one token to each of its output places. Firing of a transition is an atomic instantaneous operation, while some unspecified time can pass between enabling and firing of the transition. After the firing of transition $DSr+$ the net moves to a new marking $\{p_1, p_2\}$ and then $LDS+$ becomes enabled.

This process of moving tokens around (a.k.a. token game) in a few steps will fire transition $D-$. This leads the net into the marking $\{p_7, p_8\}$. In this marking *two* transitions $DTACK-$ and $LDS-$ become enabled. Since their input places are different they do not conflict for tokens and cannot disable each other. This represents *concurrency* between $DTACK-$ and $LDS-$. In total, there are four pairs of concurrent transitions: $(DTACK-, LDS-)$, $(DTACK-, LDTACK-)$, $(DSr+, LDS-)$, and $(DSr+, LDTACK-)$, where concurrency is a potential to fire at the same time.

1.3 State graphs

Playing the token game one can generate a *Transition System (*TS*)* – an abstract state graph in which each arc between a pair of states is labeled with the corresponding fired transition. Figure 4 depicts a TS for the READ cycle. Each state in the TS generated from a PN corresponds to a marking, which is shown at the left from the corresponding state. A TS with states labeled with markings is called a *reachability graph* of a PN. For Signal Transition Graphs each state of the corresponding TS also can be associated with a binary code of signal values, which are shown at the right from the states[1]. A TS with states labeled with binary codes of signals is called a *state graph* of an STG.

1.4 Choice and arbitration

The environment of the device has a choice to request the read or the write operation. Similarly, if an arbitration within the device is involved, then the device itself can internally make a non-deterministic choice between two requests. Choice is expressed in PNs by choice places as shown in Figure 5. Here places p_0 and p_3 are choice places, places p_1 and p_2 merge alternative branches of the behavior and all other places are removed from the figure, since they have only one input and one output arc (they are called implicit places

[1]for the sake of readability we separate with dots left handshake signals, right handshake signals, and data transceiver control signal; enabled signals are marked with an asterisk.

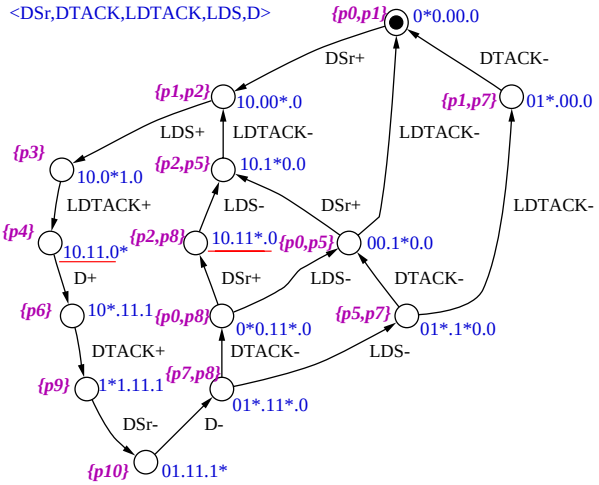


Figure 4: RG and SG for the READ cycle

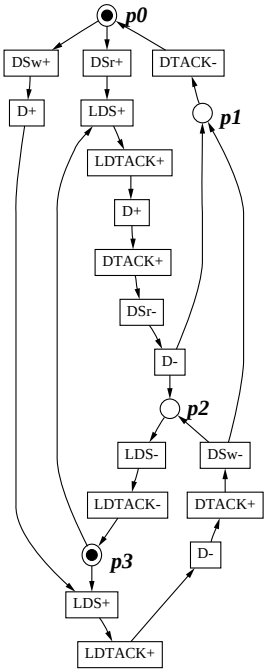


Figure 5: STG for READ and WRITE cycles

and are represented by arcs between two transitions). In the initial marking $\{p_0, p_3\}$ two input transitions are enabled, $DSw+$ and $DSr+$, but as soon as one of them fires another becomes disabled, since the token disappears from place p_0.

2 Analysis and verification

2.1 Properties

Analysis and verification are used at different stages of design.

- *Property verification.* After specifying the design it is required to check implementability properties to answer the following question: "Can the specification be implemented with an asynchronous circuit?" [14, 16]. Other properties of the specification can be of interest as well, e.g., absence of deadlocks, fairness in serving requests,

etc. General purpose verification techniques can be employed for this analysis [19].

- *Implementation verification.* After design is done fully automatically or (especially) with some manual intervention it is often desirable to check that the implementation is correct with respect to the given specification [10, 24].
- *Performance analysis and separation between events* is required (a) for determining latency and throughput of the device and (b) for logic optimization based on timing information [12, 22] (see also Section 5).

Properties required for implementability include:

- *boundedness* of the PN to guarantee that the specified state space is finite;
- *consistency* of an STG to ensure that rising and falling transitions alternate for each signal;
- *completeness of state encoding* to check that there are no conflicts in definition of Boolean functions for each non-input (i.e. output and internal) signals;
- *persistency* of the STG to verify that (a) no *non-input* signal transition can be disabled by another signal transition and (b) no *input* signal transition can be disabled by a non-input signal transition. The former ensures that no short glitches, known as *hazards*, can appear at the gate outputs, while the latter ensures that no hazards can occur at inputs of the device.

If all the above properties are satisfied, then the STG specification can be implemented as a, so-called, *speed-independent* circuit [20] [2]. Speed-independence means no hazards under any variations of gate delays if variations of some critical wire delays after forks (so-called isochronic forks) stay within reasonable bounds (e.g., within one gate delay).

Let us illustrate two of the above properties with an example. Two states in the TS in Figure 4 are underlined. They correspond to the different markings, $\{p_4\}$ and $\{p2, p8\}$, but their binary codes are equal, 10110. Moreover, enabling conditions in these two states for output signals LDS, and D are *different*. Therefore, the implied value of the next state Boolean function for signal LDS for vector 10110 should be 1 (for the first state) and 0 (for the second state). This is a conflict in the definition of the function. To resolve this conflict two methods can be employed: (a) inserting an additional state signal whose value should distinguish two conflict states or (b) concurrency reduction. In the first case one feasible solution is to insert rising transition of the additional state signal right before $LDS+$ and its falling transition right before $D-$. So conflicting states will be associated with different values of the new state signal. In the second case, a possible solution is to remove the conflicting state $\{p2, p8\}$ from the specification. The environment should usually stay untouched for the compositional reasons, therefore delaying input signals is not allowed. Hence, signal transition $DTACK-$ can be delayed until $LDS-$ fires. The automatic techniques for solving the state encoding problem are presented, e.g., in [6, 27].

To illustrate the persistency property let us consider transitions $DSw+$ and $DSr+$ in Figure 5 assuming for a moment that they are *output signals* to be implemented. Both are simultaneously enabled and disable each other after firing. Such behavior cannot be implemented without hazards unless special mutual exclusion elements (arbiters) are used.

2.2 Techniques

There are several techniques for fighting with the "state explosion problem" in analysis of Petri Net-like specifications.

- Symbolic Binary Decision Diagram-based (BDD) [3] traversal of a reachability graph allows its implicit representation which is generally much more compact than an explicit enumeration of states [24].
- Partial order reductions ([11], stubborn sets [26], identification method [14]) ignores many (or even most) of the states for analysis of certain properties.
- Structural properties of PNs (e.g., place invariants) can provide fast upper approximation of the reachability space [21, 9] and also can be used for dense variable encoding of states in the reachability graph. Structural reductions are useful as a preprocessing step in order to simplify the structure of the net before traversal or analysis, keeping all important properties.
- Unfoldings [19, 16] are finite *acyclic* prefixes of the PN behavior, representing all reachable markings. They are often more compact than the reachability graph and due to the acyclic property are well-suited for extracting ordering relations between places and transitions (concurrency, conflict and preceding). Different types of unfoldings are also used for performance analysis [12].

More details on the applicability of these techniques can be found in [13].

3 Logic Synthesis

The goal of logic synthesis is to derive a gate netlist that implements the behavior defined by the specification. For simplicity, we will illustrate this step by synthesizing a speed-independent circuit for the read cycle of the VME bus (see Figure 3).

The main steps in logic synthesis are the following:

- Encode the SG in such a way that the complete state coding property holds. This may require the addition of internal signals.
- Derive the *next-state* functions for each output and internal signal of the circuit.
- Map the functions onto a netlist of gates.

3.1 Complete State Coding

As mentioned in Section 2.1, the SG of Figure 4 has state conflicts. A possible method to solve this problem is to insert new state signals that disambiguate the encoding conflicts. Figure 6 depicts a new SG in which a new signal, *csc0*, has been inserted. Now, the next-state functions for signals LDS and D can be uniquely defined. The insertion of new signals must be done in such a way that the resulting SG preserves the properties for implementability.

3.2 Next-State Functions

When an SG fulfills all the implementability properties, a next-state function can be derived for each non-input signal.

Given a signal z, we can classify the states of the SG into four sets: positive and negative *excitation regions* ($\mathsf{ER}(z+)$ and $\mathsf{ER}(z-)$) and *quiescent regions* ($\mathsf{QR}(z+)$ and $\mathsf{QR}(z-)$).

A state belongs to $\mathsf{ER}(z+)$ if $z = 0$ and $z+$ is enabled in that state. In this situation, the value of the signal is denoted by 0^* in the SG. A state belongs to $\mathsf{QR}(z+)$ if s in stable 1 state. These definitions are analogous for $\mathsf{ER}(z-)$ and $\mathsf{QR}(z-)$.

The next-state function for a signal z is defined as follows:

$$f_z(s) = \begin{cases} 1 & \text{if } s \in \mathsf{ER}(z+) \cup \mathsf{QR}(z+) \\ 0 & \text{if } s \in \mathsf{ER}(z-) \cup \mathsf{QR}(z-) \\ - & \text{otherwise} \end{cases}$$

[2] Also called *quasi-delay-insensitive* in the literature [18, 2]

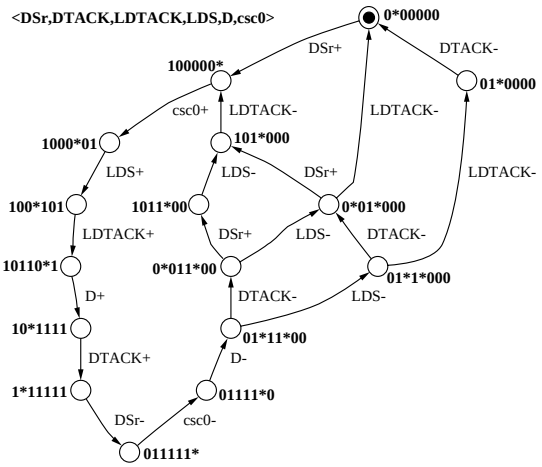


Figure 6: SG for the READ cycle with complete state coding.

where s denotes the binary code of a state. The fact that $f_z(s) = -$ indicates that there is no state with such code in the SG and, thus, s can be considered as a *don't care* condition for boolean minimization.

Once the next-state function has been derived, boolean minimization can be performed to obtain a logic equation that implements the behavior of the signal. In this step it is crucial to make an efficient use of the don't care conditions derived from those binary codes not corresponding to any state of the SG. For the example of Figure 6, the following equations can be obtained:

$$D = LDTACK \cdot csc0; \quad LDS = D + csc0$$
$$DTACK = D; \quad csc0 = DSr \cdot (csc0 + \overline{LDTACK})$$

A well known result in the theory of asynchronous circuits is that any circuit implementing the next-state function of each signal with only one atomic complex gate is speed independent. By atomic gate we mean a gate without internal hazardous behavior [14, 17]. Two possible hazard-free gate mappings for the next-state function of the READ cycle example are shown in Figure 7,a and b.

However, there could be two obstacles in the actual implementation of the next state functions: (a) a logic function can be too complex to be mapped into one gate available in the library; (b) the solution requires the use of gates which are not typically present in standard synchronous libraries. The second is the case with solution Figure 7,a. A gate pictured as a circle with "C" is a so-called C-element [20]: a popular asynchronous latch with the next state function $c = ab + c(a + b)$. Its output, c, goes high (low) if both inputs, a and b, go high (low); otherwise, it keeps the previous value.

3.3 Hazards

A crucial problem which makes solution of logic decomposition problem for asynchronous design difficult is a problem of *hazards* [25, 23]. Recent development in [23] shows that if the so-called *Fundamental mode* is acceptable (input cannot change until all internal circuit activity stabilizes), then most of the known methods of logic minimization can be gracefully extended to asynchronous hazard-free minimization. These results can further be extended to FSMs [29].

Unfortunately, the Fundamental mode is often too restrictive and in particular is not satisfied for logic implementing signal functions in synthesis using STGs.

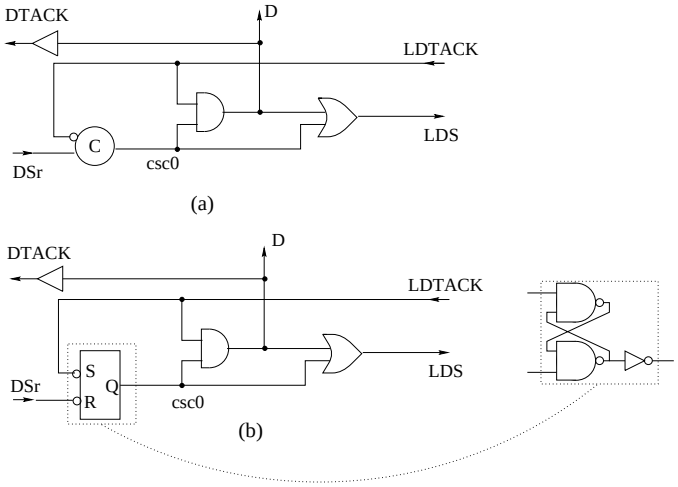


Figure 7: Implementations with latches

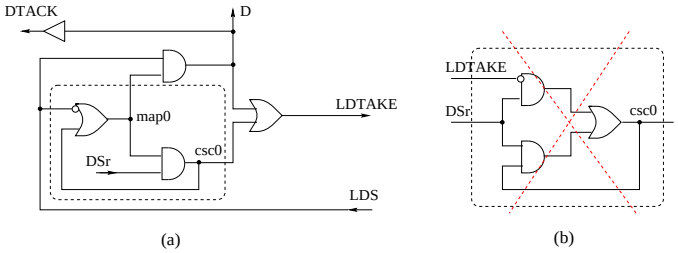


Figure 8: Implementation with two-input gates

3.4 Decomposition and Technology Mapping

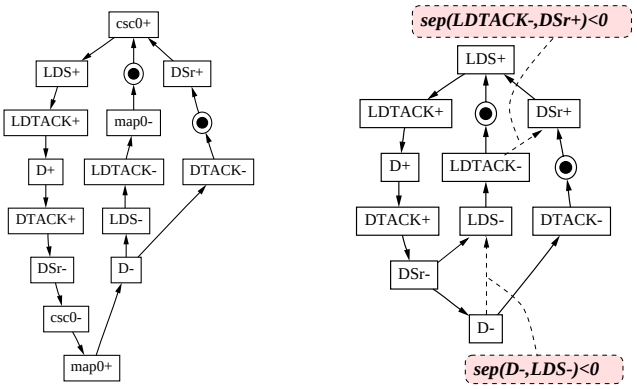
One of the partial solutions to the logic decomposition for non-fundamental mode, called the monotonous cover requirement [1, 15], allows one to decompose any function into two-level combinational logic and a latch. This does not solve however a problem of breaking gates if the fan-in or fan-out is too large. The latest results [4, 5] allow one to obtain a hazard-free decomposition (and then map the decomposed solution into the available library) without [4] or with [5] gate sharing into gates with restricted fan-in.

Applying method from [5] two other correct solutions can be found for mapping the control for READ cycle into two inputs gate library: solution in Figure 7,b uses a standard reset dominant RS-latch instead of the C-element; solution in Figure 8,a uses *only* combinational gates. This solution seems to be a standard synchronous decomposition for the function of signal $csc0 = DSr \cdot (csc0 + \overline{LDTACK})$:

$$map0 = csc0 + \overline{LDTACK}; \quad csc0 = DSr \cdot map0$$

Note, however, that signal $map0$ is also fed to gate $D = LDS \cdot map0$. It is only because of this multiple acknowledgment of $map0$ by two different gates, that this solution for the READ cycle control is hazard-free: every rising transition at $map0$ is acknowledged by signal D, while every falling transition – by signal $csc0$. Another synchronous decomposition for $csc0$ presented in Figure 8,b is hazardous and cannot be accepted.

The technique for decomposition and technology mapping from [5] is based on using candidates for decomposition extracted by algebraic factorization and boolean relations and inserting hazard-free signals with multiple acknowledgment.



(a) (b)

Figure 9: (a) STG extracted for the two-input combinational gate circuit, (b) timing STG with separation constraints for the optimized circuit

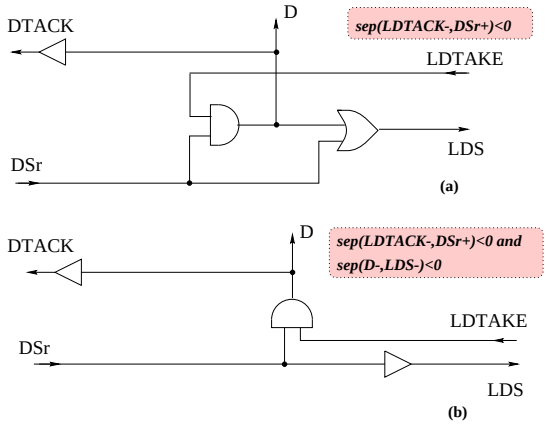


Figure 10: Circuits for the READ cycle after timing optimization

4 Back annotation

State regions [8] are sets of states such that they correspond to a place (*regions*) or a transition of the PN (*excitation regions*). Entry and exit arcs for a region correspond to input and output transitions of a place. Apart from being useful for state exploration regions provide another important feature: at any step of the design process a PN corresponding to the current TS can be extracted and back-annotated to the designer. This is useful both for interactions with the design process and for the performance and timing analysis of the circuit. An example of a PN extraction is shown in Figure 9,a.

5 Timing Optimization

The power of optimization based on timing information is two-fold.

- Timing constraints always *reduce* the set of reachable states and hence increase the number of don't care states [22]. Moreover this concurrency reduction does not introduce new dependencies between signals since it is fully based on timing not on logic ordering.
- Using timing requirements it is possible to *extend* the set of states in which signal is enabled *without* changing the set of reachable states: signal transition enabling does not cause signal firing if other enabling signals are known to be (or can be made) faster.

Let us illustrate how timing information can increase the flexibility in logic optimization by example of the READ cycle. Assume first that, as a part of the initial specification, it is given that the reset at the right side handshake is always faster than the next read request at the left side handshake, formally: maximal separation[12] between transitions $LDTACK-$ and $DSr+$ is negative, $Sep(LDTACK-, DSr+) < 0$. Then there is no need in the additional state encoding signal and the circuit is simplified to Figure 10,a.

Assume next that the physical design level tools achieve control over the delay information using gate and transistor sizing, placement and routing, and constraining interconnect delays. Then the logic-level synthesis tools can perform logic optimization at the same time generating separation constraints that must be implemented by the physical level tools. For example, it is possible to start enabling of $LDS-$ right after $DSr-$ instead of $D-$ given that the requirement $Sep(D-, LDS-) < 0$ will be satisfied. This requirement is satisfied if the maximal delay of $D-$ is smaller than the minimal possible delay of $LDS-$ that can be implemented, e.g., by transistor sizing or delay padding. The resulting circuit corresponding to both timing requirements is shown in Figure 10,b. Back-annotation to an extended PN with relational timing constraints (so-called *lazy* PNs) can be done for the circuits optimized based on timing information (see Figure 9,b).

6 Other Design Techniques

This paper has presented a design methodology based on Petri net specifications of the behavior of a circuit. However, other models have been proposed in the literature. Among them, we can point up the methods based of burst-mode machines [29] and on syntax-directed [2] or transformation-based [18] translation from process algebras.

Burst-mode machines work under the so-called *fundamental mode* assumption, i.e. after each burst of inputs events accepted by the system, the environment allows the circuit to stabilize before reacting to the output events. This assumption is realistic for many applications and enables the utilization of combinational logic minimization methods for synchronous circuits with ad-hoc extensions to prevent hazardous behavior [23].

Translation from process algebras has been proposed for formalisms derived from CSP. Syntax-directed translation derives a netlist of components that implement the behavior of each of the constructs of the language (parallel/sequential composition, choice, communication, synchronization, etc.). The size of the resulting circuit is linearly dependent on the size of the input description. This fact enables designers and tools to predict the circuit's performance and complexity parameters at the earliest steps of the design process.

Other efforts have been devoted to map asynchronous specifications into standard HDLs aiming at the simulation and validation with commercial tools [28].

7 Summary

In the last few years, the techniques for asynchronous designed have matured. Among the applications for asynchronous design we can point up asynchronous interfaces, high-performance computing, low-power and low-emission design, etc. There are also applications at the system level, e.g. hardware-software co-design.

Recently there has been an increasing interest of few but large-scale industries (e.g. Intel, Philips, Sharp, ARM, Cogency, SUN, HP) in asynchronous design targeting at different goals: low power, high performance, etc.

Asynchrony introduces a new paradigm in logic design. Asynchronous circuits are much more difficult to design and, for this reason, it is crucial to provide CAD tools to handle the most difficult tasks automatically. Most of the steps of the design process presented in this tutorial are supported by the tool `petrify` available at **DAC paper home URL: `http://www.lsi.upc.es/~jordic/petrify`**.

For a more complete tutorial in PN-based design of asynchronous control circuits we refer to [7]. For further information on asynchronous design, the reader can look at the Asynchronous Logic Home Page (`http://www.cs.man.ac.uk/amulet/async/index.html`) and the proceedings of the ASYNC Symposiums.

An extended version of this paper can be found in [13].

Acknowledgments

We wish to thank Luciano Lavagno, Alexander Taubin, and Alex Yakovlev for numerous discussions on the topics presented in this paper.

References

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

[2] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[3] Randal Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[4] S. Burns. General conditions for the decomposition of state holding elements. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Aizu, Japan*, March 1996.

[5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, E. Pastor, and A. Yakovlev. Decomposition and technology mapping of speed-independent circuits using boolean relations. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, November 1997.

[6] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. A region-based theory for state assignment in speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 16(8):793–812, August 1997.

[7] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Synthesis of control circuits from STG specifications. In handouts of the Summer School on Asynchronous Circuit Design, August 1997. `http://www.lsi.upc.es/~jordic/petrify/refs/summer97.ps.gz`.

[8] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing Petri nets from state-based models. In *Proceedings of the International Conference on Computer-Aided Design*, pages 164–171, November 1995.

[9] J. Desel and J. Esparza. *Free-choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.

[10] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[11] P. Godefroid. Using partial orders to improve automatic verification methods. In E.M Clarke and R.P. Kurshan, editors, *Proc. International Workshop on Computer Aided Verification*, 1990. DIMACS Series in Discrete Mathematica and Theoretical Computer Science, 1991, pages 321-340.

[12] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrenct systems. *IEEE Transactions on Computers*, 44(11):1306–1317, November 1995.

[13] M. Kishinevsky, J. Cortadella, A. Kondratyev, and L. Lavagno. Asynchronous interface specification, analysis and synthesis. Technical Report LSI-98-14-R, Technical University of Catalonia, March 1998. `http://www.lsi.upc.es/dept/techreps/ps/R98-14.ps.gz`.

[14] M. A. Kishinevsky, A. Y. Kondratyev, A. R. Taubin, and V. I. Varshavsky. *Concurrent Hardware. The Theory and Practice of Self-Timed Design*. John Wiley and Sons Ltd., 1994.

[15] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *Proceedings of the Design Automation Conference*, pages 56–62, June 1994.

[16] A. Kondratyev, M. Kishinevsky, A. Taubin, and S. Ten. Analysis of Petri nets by ordering relations in reduced unfoldings. *Formal Methods in System Design*, 12(1):5–38, 1997.

[17] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for synthesis and testing of asynchronous circuits*. Kluwer Academic Publishers, 1993.

[18] A. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communications*, The UT Year of Programming Series. Addison-Wesley, 1990.

[19] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[20] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[21] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, April 1989.

[22] Chris J. Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[23] Steven M. Nowick and David L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Transactions on Computer-Aided Design*, 14(8):986–997, August 1995.

[24] Oriol Roig, Jordi Cortadella, and Enric Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *16th International Conference on the Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 374–391, 1995.

[25] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[26] Antti Valmari. Stubborn sets for reduced state space generation. *Lecture Notes in Computer Science; Advances in Petri Nets 1990*, 483:491–515, 1991.

[27] P. Vanbekbergen, B. Lin, G. Goossens, and H. De Man. A generalized state assignment theory for transformations on Signal Transition Graphs. *Journal of VLSI Signal Processing*, 7(1-2):101–116, 1994.

[28] Peter Vanbekbergen, Albert Wand, and Kurt Keutzer. A design and validation system for asynchronous circuits. In *Proc. ACM/IEEE Design Automation Conference*, June 1995.

[29] K. Y. Yun and D. L. Dill. Automatic synthesis of 3D asynchronous state machines. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.