Hardware and Petri Nets: Application to Asynchronous Circuit Design

J. Cortadella[1], M. Kishinevsky[2], A. Kondratyev[3], L. Lavagno[4], and A. Yakovlev[5]

[1] Universitat Politècnica de Catalunya, Department of Software, Campus Nord, Mòdul C6, 08034 Barcelona, Spain.
jordic@lsi.upc.es
[2] Intel Corporation, JFT-104, 2111 N.E. 25th Ave., Hillsboro, OR 97124-5961, USA.
mkishine@ichips.intel.com
[3] Theseus Logic, 710 Lakeway Drive, suite 230, Sunnyvale, CA 94087, USA.
alex.kondratyev@theseus.com
[4] Universitá di Udine, DIEGM, Via delle Scienze 208, I-33100 Udine, Italy.
lavagno@uniud.it
[5] University of Newcastle upon Tyne, Department of Computing Science, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, UK.
Alex.Yakovlev@newcastle.ac.uk

Abstract. Asynchronous circuits is a discipline in which the theory of concurrency is applied to hardware design. This paper presents an overview of a design framework in which Petri nets are used as the main behavioral model for specification. Techniques for synthesis, analysis and formal verification of asynchronous circuits are reviewed and discussed.

1 Introduction

Finite State Machines has been the most traditional model of computation for sequential circuits [25, 26]. It is a state-based model in which the system, being in a state, reads some inputs, writes some outputs and moves to another state. Time is discretized by the notion of cycle, which is the time that takes the system to move from one state to another. This model is appropriate to derive circuit implementations with a periodic signal, the *clock*, that dictates the time instants in which the system changes state. The cycle is the finest degree of granularity at which operations are scheduled. Thus, two operations are concurrent if they are scheduled at the same cycle. The cycle delay is determined by the worst-case delay the circuit takes to perform the operations scheduled at any of the cycles. Synchronization among operations is implicit, i.e. the initiation of an operation always starts at a clock edge and completes after a fixed quantity of cycles at another clock edge. Thus, clock edges indicate the *initiation and completion of several actions* simultaneously.

After a long period of hibernation, asynchronous circuits woke up fifteen years ago as a potential solution to some of the design problems posed by VLSI technologies [10]. In asynchronous circuits, the sequencing of operations is no longer dictated by a clock, but by *events* that indicate the *initiation and completion of individual actions*. The correctness of an asynchronous circuit not only depends on its structure, but also on the timing behavior of the individual gates and their interaction. Thus, a circuit can be modeled as a set of processes (gates) that communicate through channels (wires) and modify the state of the system represented by a set of Boolean variables (signals). A gate is *enabled* when the value of its output is different from the value calculated by its logic function. An enabled gate can produce a transition (event) on the output signal by changing its value.

The view of an asynchronous circuit as a concurrent system makes event-based models of computation more appropriate for analysis and synthesis. For this reason, process algebras, such as CSP [16], and Petri nets [34] have raised the interest of the researchers in this discipline.

This paper presents an overview of a design methodology for asynchronous circuits that uses Petri nets as the underlying model for specification, synthesis and verification. Section 2 explains how the behavior of an asynchronous circuit can be specified by using Petri nets. Section 3 presents a set of sufficient properties for a specification to be implementable as a speed-independent circuit. The techniques to derive an implementation with logic gates are described in Sect. 4. The retrieval of the actual circuit behavior as a Petri net is known as back-annotation and is presented in Sect. 5. Finally, different strategies to fight against the state explosion problem in analysis and formal verification are reviewed in Sect. 6.

2 Timing diagrams, Petri nets and Signal Transition Graphs

For most circuit designers, Petri nets resemble timing diagrams, a model to specify asynchronous interfaces as signal waveforms that explictly indicate the causality and concurrency relations among signal transitions.

Figure 1(a) depicts the block diagram of a VME bus controller. According to its functionality, the controller has three sets of "handshake" signals: those interacting with the bus (DSr, DSw and $DTACK$), those interacting with the device connected to the bus (LDS and $LDTACK$), and that controlling the transceiver that connects the bus with the device (D).

The behavior of the controller is as follows: a request to read from or write into 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 1(b) shows the timing diagram of the READ cycle.

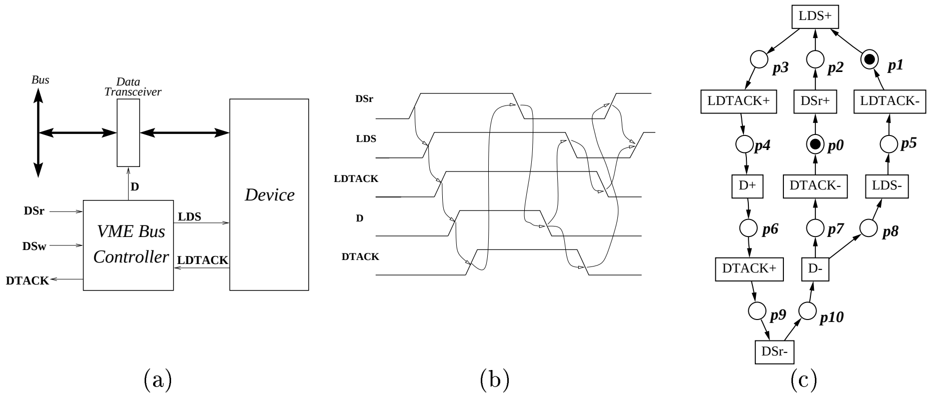


Fig. 1. (a) VME bus controller, (b) READ cycle, (c) Signal Transition Graph.

Figure 1(c) is a Petri net describing the same behavior. The events are interpreted as rising (+) and falling (−) signal transitions. Petri Nets with such interpretation are called *Signal Transition Graphs* (STG) [5, 38]. Three types of signals can be distinguished in an STG: input, output and internal. The behavior of the input signals (DSr, DSw and $LDTACK$) is determined by the environment. The behavior of the output (D, $DTACK$ and LDS) and internal signals is determined by the system and is the one that must be implemented by the circuit. Typically, internal signals are incorporated during the synthesis of the circuit to solve some implementation problems (encoding, decomposition) and do not appear in the original specification of the system.

Figure 2 depicts the STG that describes the complete behavior of the controller[1]. Unlike timing diagrams, STGs can specify choice and non-determinism. In this example, the initial marking models the situation in which the environment can non-deterministically choose to initiate a read cycle by firing $DSr+$, or a write cycle by firing $DSw+$.

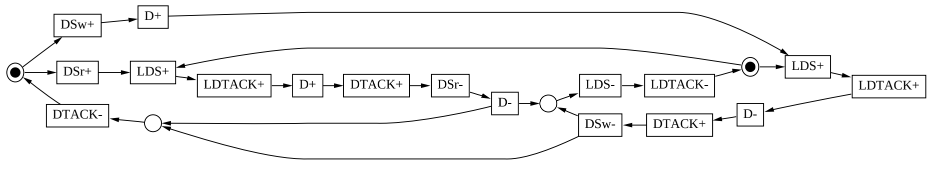


Fig. 2. STG for the READ and WRITE cycles.

[1] Usually, places with only one predecessor and one successor transition are not explicitly drawn in STGs

This paper presents a methodology to synthesize asynchronous circuits from STGs. This methodology has been completely automated and implemented in a synthesis tool called `petrify` [35]. The specification of the READ cycle shown in Fig. 1(c) will be used as an example to illustrate this methodology along the paper. For the sake of simplicity, the WRITE cycle will be ignored.

3 Implementability properties

The goal of the synthesis methodology is to derive a *speed-independent* circuit that realizes the specified behavior. Speed independence is a property that guarantees a correct behavior under the assumption that all gates have an unbounded delay and all wires have a negligible delay [28].

A specification must fulfil certain properties to be implementable as a speed-independent circuit. These properties can be better described on the state graph of the specification.

3.1 State Graph

The state graph (SG) of a specification is the transition system obtained from the reachability analysis of an STG. Each state corresponds to a marking of the STG and each arc corresponds to the firing of a signal transition. Figure 3 shows the SG of the read cycle specified in Fig. 1(c).

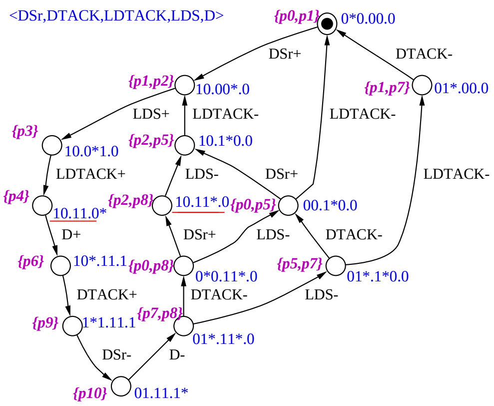


Fig. 3. State graph for the READ cycle.

In the SG, each state is assigned a binary vector with the value of all signals at that state. For the sake of readability, the control signals corresponding to the left handshake, right handshake and data transceiver are separated by dots. Enabled signals are marked with an asterisk. For example, the state corresponding to the marking $\{p_7, p_8\}$ has the code $01^*.11^*.0$, indicating that DSr and D are at 0, and

$DTACK$, $LDTACK$ and LDS are at 1. Moreover, signals $DTACK$ and LDS are both enabled, i.e. ready to change their value. The initial state corresponds to the marking $\{p_0, p_1\}$ and all signals at 0.

3.2 Properties for implementability

The properties required for the specification to be implementable as a speed-independent circuit are the following:

- *Boundedness* of the STG that guarantees the SG to be finite.
- *Consistency* of the STG, that consists in ensuring that the rising and falling transitions of each signal alternate in all possible runs of the specification.
- *Completeness of state encoding* that ensures that there are no two different states with the same signal encoding and different behavior of the output or internal signals.
- *Persistency* of signal transitions in such a way that no signal transition can be disabled by another signal transition, unless both signals are inputs. This property ensures that no short glitches, known as *hazards*, will appear at the disabled signals.

The SG of Fig. 3 fulfils boundedness, consistency and persistency. However, it does not have completeness of state encoding. The states corresponding to the markings $\{p_4\}$ and $\{p_2, p_8\}$ have the same code. Moreover, the behavior of the output signals in those states is different. In the state $\{p_4\}$, the event $D+$ is enabled, whereas in the state $\{p_2, p_8\}$, the event $LDS-$ is enabled. Intuitively, this means that the information provided by the value of the signals is not enough to determine the future behavior of the system. This will result in an ambiguity in the definition of the next-state logic functions.

4 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 Fig. 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.

4.1 Complete State Coding

As mentioned in Sect. 3.2, the SG of Fig. 3 has state conflicts. A possible method to solve this problem is to insert new state signals that disambiguate the encoding conflicts. Figure 4 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.

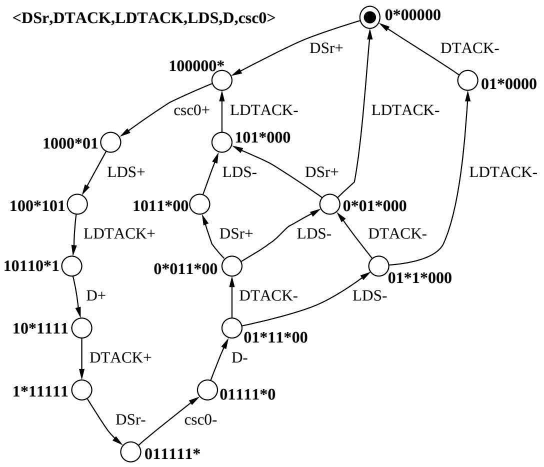


Fig. 4. SG for the READ cycle with complete state coding.

4.2 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 is 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:

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

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 Fig. 4, the following equations can be obtained:

$$\begin{array}{ll} D = LDTACK \ \wedge \ csc0; & LDS = D \ \vee \ csc0 \\ DTACK = D; & csc0 = DSr \ \wedge \ (csc0 \ \vee \ \neg LDTACK) \end{array}$$

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 [18, 22]. Two possible hazard-free gate mappings for the next-state function of the READ cycle example are shown in Fig. 5(a) and 5(b).

However, there could be two obstacles in the actual implementation of the next state functions:

- a logic function can be too complex to be mapped into one gate available in the library,
- the solution requires the use of gates which are not typically present in standard synchronous libraries

The second is the case with the solution in Fig. 5(a). A gate drawn as a circle with "C"is the so-called C-element [27]: a popular asynchronous latch with the next state function $c = (a \wedge b) \vee (c \wedge (a \vee b))$. Its output, c, goes high (low) if both inputs, a and b, go high (low), otherwise it keeps the previous value.

Decomposing the circuit into a set of simpler gates that can be implemented in a given technology is a problem that has been studied by several authors [2, 4, 7, 20].

4.3 Size of the state space

The derivation of Boolean equations from a specification requires to calculate the encoding of all the states of the system. Unfortunately, the size of the state space of a concurrent system can be exponential on the size of the specification.

The existing tools for synthesis of asynchronous circuits use different methods to fight against the state explosion problem. In *Burst-mode automata* concurrency is restricted in such a way that bursts of input and output events are serialized [31]. Thus, the behavior can be represented by a Mealy-like automata with a manageable number of states in which the concurrency is annotated as input/output bursts on the arcs of the automata.

When no constraints are imposed on the type of concurrency manifested by the system, the knowledge and manipulation of the state space usually becomes the dominant part on the complexity of the synthesis algorithms. This is the case when Petri nets are used as the specification formalism. Different strategies have been used to calculate the state space:

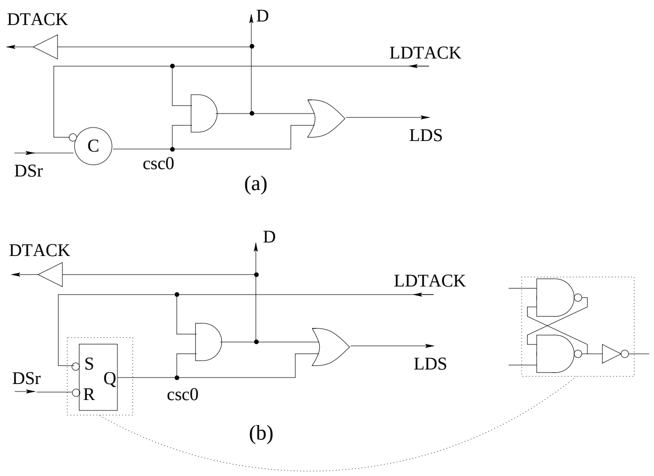


Fig. 5. Implementations with 2-input gates

– Restrict the specification to a certain class of Petri nets, e.g. Free-choice Petri nets [11], and use structural methods that can manipulate the state space by only analyzing the underlying graph of the net and without explicitly generating the states [33]. Techniques for the efficient calculation of concurrency relations are crucial in this context [21].
– Annotate timing on the events, e.g. min/max firing delays, and calculate the reduced state space reachable only under the specified timing constraints. This is the approach used for the synthesis of timed asynchronous circuits [30].
– Use symbolic methods, such as Binary Decision Diagrams [3], to implicitly represent and manipulate the complete state space. This is the approach used in the synthesis tool `petrify` [35].

5 Back-annotation

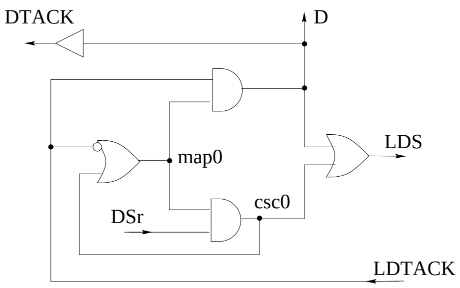
As important as the structure of the circuit resulting from the synthesis of the specification is the actual behavior of the circuit. During synthesis, internal signals might have been added to encode the states and decompose complex gates.

On the other hand, and although paradoxical, reducing concurrency [8] is one of the proposed approaches to improve the efficiency of the final circuit. Reducing concurrency directly results in a reduction of the state space and, thus, in an increase of the don't care conditions for logic minimization. In general, concurrency reduction produces smaller circuits, but it may also produce faster circuits: the system manifests less concurrency but the events take less time to fire.

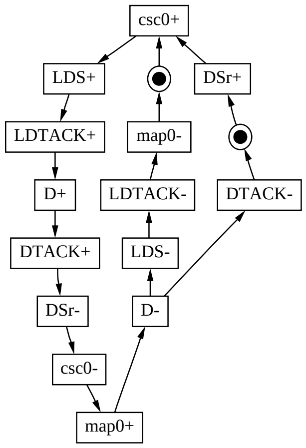
In the synthesis flow, signal insertion and concurrency reduction are usually performed at the level of SG. Providing a behavioral description of the synthesized circuit with the same formalism used for specification, e.g. Petri nets, allows the designer to easily interact with the synthesis framework and manually introduce those optimizations that automatic tools cannot find.

The problem of deriving a Petri net from a transition system was first tackled in [14] and studied by other authors [1, 12]. In these works, the theory of regions was developed to characterize the class of transition systems which correspond to elementary Petri nets. That work was extended in [9] to propose algorithms for the synthesis of safe Petri nets from any finite transition system. This work was crucial to provide the synthesis tool `petrify` [35] the capability of deriving a succint behavioral description of the synthesized circuits.

An example of back-annotation is presented in Fig. 6. The circuit is an implementation of the READ cycle specified in Fig. 1(a). In this implementation, only combinational 2-input gates have been used. With respect to the original specification, two new signals have been incorporated: *csc0* to uniquely encode the states, as shown in Fig. 4, and *map0* to decompose a complex gate into smaller gates. By using the theory of regions, the Petri net of Fig. 6(b) is automatically synthesized and shown to the designer for analysis of the circuit's behavior.



(a)



(b)

Fig. 6. (a) A 2-input gate circuit with no latches, (b) circuit behavior.

6 Analysis and formal verification

Analysis and formal verification are used at different stages of the design flow of asynchronous circuits. In particular for:

– *Property verification.* After specifying the design it is required to check implementability properties to answer the following question: "Can the specifi-

cation be implemented with an asynchronous circuit?" [18, 19]. 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 [23].
– *Implementation verification.* After the design has been done fully automatically or with some manual intervention it is often desirable to check that the implementation conforms the given specification [13, 37].
– *Performance analysis and separation between events* is required (a) for determining the latency and throughput of the circuit and (b) for logic optimization based on timing information [17, 30].

6.1 Techniques

As mentioned in Sect. 4.3, the state space of a concurrent specification is one of the major bottlenecks for the analysis of this type of systems. Here we present some techniques that have been succesfully applied in the area of formal verification of asynchronous circuits.

– 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 [37].
– Partial order reductions ([15], stubborn sets [39], identification method [18]) can abstract and ignore many of the irrelevant states for the verification of certain properties.
– Structural properties of Petri nets (e.g., place invariants) can provide fast upper approximation of the reachability space [6, 11, 29] and can be also 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, 24] are finite *acyclic* prefixes of the Petri net behavior, representing all reachable markings. They are often more compact than the reachability graph and well-suited for extracting ordering relations between places and transitions (concurrency, conflict and precedence). Different types of unfoldings are also used for performance analysis [17].

As an example, we next illustrate how structural properties of a Petri net and BDD-based representations can be combined for an efficient analysis of the state space. Figure 7 is the result of applying linear reductions to the STG from Fig. 2. Using more elaborate reductions (place and transition fusions) it is possible to reduce the whole Petri net from Fig. 1(c) to a single self-loop transition [29].

The BDD-based method used for deriving the transition function and calculating the reachable markings of a Petri net are similar to those used for reachability analysis and equivalence checking of finite state machines [23]: starting from the initial marking and by iteratively applying the transition relation until a fix point is reached, the characteristic function of the reachability set is

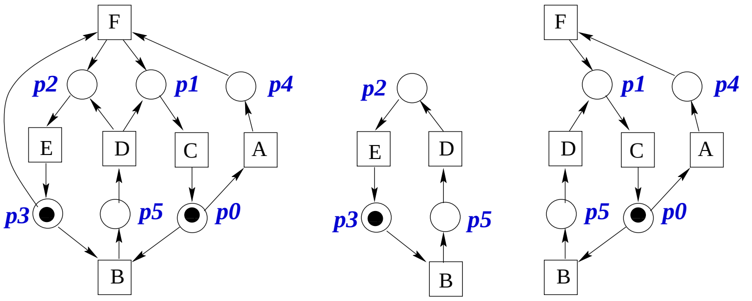


Fig. 7. STG after linear reduction and two state machine components

calculated. Under the assumption that the Petri net is safe, one could use a naive encoding, such as one Boolean variable per place, to encode the markings. However, this may be too costly for large specifications.

The following observation can be made: the sets of places $P_0 = \{p_2, p_3, p_5\}$ and $P_1 = \{p_0, p_1, p_4, p_5\}$ of the Petri net in Fig. 7 define two state machines [11, 29] with the sets of transitions $T_0 = \{B, D, E\}$ and $T_1 = \{A, B, C, D, F\}$, respectively. This information can be structurally obtained by using algebraic methods. State machines (see Fig. 7) correspond to place-invariants of the Petri net and preserve their token count in all reachable markings. Therefore, the following are two invariants for the net:

$$\begin{aligned} I_1(p_2, p_3, p_5): \quad & M(p_2) + M(p_3) + M(p_5) = 1 \\ I_2(p_0, p_1, p_4, p_5): \quad & M(p_0) + M(p_1) + M(p_4) + M(p_5) = 1 \end{aligned}$$

If invariants $I_1(p_2, p_3, p_5)$ and $I_2(p_0, p_1, p_4, p_5)$ are represented as Boolean functions (e.g., by using BDDs), then the conjunction on these two functions give, for this example, the characteristic function of the exact reachability set of markings. In general a conjunction of any set of invariants gives an *upper approximation* of the reachability set, which is useful for conservative verification.

On the other hand, and by using the previous invariants, a dense encoding for places with the vector of Boolean variables $V = (v_0, v_1, v_2, v_3)$ can be proposed (see Table 1).

Table 1. Encoding for the places and reachable markings of the Petri net in Fig. 7.

place	v_0	v_1	v_2	v_3	χ_p
p_2	0	0	-	-	$\overline{v_0}\,\overline{v_1}$
p_3	0	1	-	-	$\overline{v_0}v_1$
p_5	1	-	1	1	$v_0v_2v_3$
p_0	-	-	0	0	$\overline{v_2}\,\overline{v_3}$
p_1	-	-	0	1	$\overline{v_2}v_3$
p_4	-	-	1	0	$v_2\overline{v_3}$

marking	χ_M
$\{p_0, p_3\}$	$\overline{v_0}\,v_1\overline{v_2}\,\overline{v_3}$
$\{p_5\}$	$v_0v_2v_3$
$\{p_1, p_2\}$	$\overline{v_0}\,\overline{v_1}\,\overline{v_2}v_3$
$\{p_1, p_3\}$	$\overline{v_0}v_1\overline{v_2}v_3$
$\{p_0, p_2\}$	$\overline{v_0}\,\overline{v_1}\,\overline{v_2}\,\overline{v_3}$
$\{p_3, p_4\}$	$\overline{v_0}v_1v_2\overline{v_3}$
$\{p_2, p_4\}$	$\overline{v_0}\,\overline{v_1}v_2\overline{v_3}$

χ_p is the characteristic function of a place and represents the those markings that have a token in place p, whereas χ_M is the characteristic function of marking M. The set of variables v_0 and v_1 is used to encode the places involved in the invariant I_1. Similarly, v_2 and v_3 are used to encode the places involved in the invariant I_2. Place p_5 supports both invariants and is encoded with variables from both sets. The characteristic function of all reachable markings, $R(V)$, can be represented by a single Boolean equation obtained as the disjunction of the characteristic functions of the markings. When simplified, the equation can be represented as follows:

$$R(V) = \overline{v_0}(\overline{v_2} \vee \overline{v_3}) \vee v_0 v_2 v_3 = v_0 \Leftrightarrow v_2 v_3.$$

With $R(V)$ it is possible to verify properties of the net. For example, one might what to verify that transitions D and F are not concurrent[2]. This can be verified by proving that there is no reachable marking in which both transitions are enabled. The characteristic function of D and F being enabled can be represented as:

$$\text{Enabled}(D) \wedge \text{Enabled}(F) = \chi_{p_5} \wedge (\chi_{p_3} \wedge \chi_{p_4})$$

Therefore, the characteristic function of the markings in which both transitions are enabled is

$$R(V) \wedge \text{Enabled}(D) \wedge \text{Enabled}(F).$$

It can be easily observed that this logic proposition is a contradiction, thus proving that there is no reachable marking in which D and F are enabled.

This type of techniques has been succesfully applied to verify concurrent systems specified with Petri nets [32].

7 Conclusions

Event-based models for concurrency are being used for the specification, synthesis, analysis and verification of asynchronous circuits. Among them, Petri nets seem to be the most appropriate model that offers the following features:

- A succint representation of the behavior of concurrent systems with no restriction on the type of allowed concurrency among events.
- A easy calculation and manipulation of the state space, thus enabling the use of efficient techniques for state encoding and logic synthesis.
- An intermediate representation for higher-level formalisms like process algebras.

[2] This property can be easily verified by using algebraic methods. Here, we only want to illustrate the mechanics of formal verification when using symbolic encoding techniques.

Asynchronous circuit design is still in its prehistory. In the future, we foresee an increasing interest in such type of circuits and design methodologies. High-level synthesis tools will be constructed and used in a broader set of applications. For this reason, techniques to synthesize and verify highly-complex concurrent systems will be required. This is an area where theoreticians on concurrency have a chance to apply their findings and collect new challenging problems to solve.

Acknowledgements. This work has been partially funded by the ACiD-WG (ESPRIT 21949), a grant by Intel Corporation, CICYT TIC 98-0410, CICYT TIC 98-0949 and CIRIT 1999SGR-150.

References

1. E. Badouel and Ph. Darondeau. Theory of regions. In *[36]*, pages 529–586. Springer-Verlag, 1998.
2. P. Beerel and T.H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, November 1992.
3. R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
4. S. M. Burns. General condition for the decomposition of state holding elements. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.
5. T.-A. Chu and L. A. Glasser. Synthesis of self-timed control circuits form graphs: An example. In *Proc. International Conf. Computer Design (ICCD)*, pages 565–571. IEEE Computer Society Press, 1986.
6. J. Cortadella. Combining structural and symbolic methods for the verification of concurrent systems. In *Proc. of the International Conference on Application of Concurrency to System Design*, pages 2–7, March 1998.
7. Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor, and Alexandre Yakovlev. Decomposition and technology mapping of speed-independent circuits using Boolean relations. *IEEE Transactions on Computer-Aided Design*, 18(9), September 1999.
8. Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Automatic handshake expansion and reshuffling using concurrency reduction. In *Proc. of the Workshop Hardware Design and Petri Nets (within the International Conference on Application and Theory of Petri Nets)*, pages 86–110, June 1998.
9. Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexandre Yakovlev. Deriving Petri nets from finite transition systems. *IEEE Transactions on Computers*, 47(8):859–882, August 1998.
10. Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. In A. Kent and J. G. Williams, editors, *The Encyclopedia of Computer Science and Technology*, volume 38. Marcel Dekker, New York, February 1998.
11. J. Desel and J. Esparza. *Free-choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1995.

12. J. Desel and W. Reisig. The synthesis problem of Petri nets. *Acta Informatica*, 33(4):297–315, 1996.
13. David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.
14. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures. Part I, II. *Acta Informatica*, 27:315–368, 1990.
15. 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.
16. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
17. H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE Transactions on Computers*, 44(11):1306–1317, November 1995.
18. Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.
19. Alex Kondratyev, Michael Kishinevsky, Alexander Taubin, and Sergei Ten. Analysis of Petri nets by ordering relations in reduced unfoldings. *Formal Methods in System Design*, 12(1):5–38, January 1998.
20. Alex Kondratyev, Michael Kishinevsky, and Alex Yakovlev. Hazard-free implementation of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 17(9):749–771, September 1998.
21. A. Kovalyov. A Polynomial Algorithm to Compute the Concurrency Relation of a Regular STG. In A. Yakovlev, L. Gomesa, and L. Lavagno, editors, *Hardware Design and Petri Nets*, pages 107–126. Kluwer Academic Publishers, March 2000.
22. Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.
23. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
24. K. L. McMillan. Trace theoretic verification of asynchronous circuits using unfoldings. In *Proc. International Workshop on Computer Aided Verification*, 1995.
25. G.H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical J.*, 34(5):1045–1079, 1955.
26. E.F. Moore. Gedanken experiments on sequential machines. *Automata Studies*, pages 129–153, 1956.
27. David E. Muller. Asynchronous logics and application to information processing. In *Symposium on the Application of Switching Theory to Space Technology*, pages 289–297. Stanford University Press, 1962.
28. 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.
29. T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, April 1989.
30. Chris J. Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.
31. S. M. Nowick and B. Coates. Automated design of high-performance asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)*. IEEE Computer Society Press, October 1994.
32. E. Pastor, J. Cortadella, and M.A. Peña. Structural methods to improve the symbolic analysis of Petri nets. In *Application and Theory of Petri Nets 1999*, Lecture Notes in Computer Science, June 1999.

33. Enric Pastor, Jordi Cortadella, Alex Kondratyev, and Oriol Roig. Structural methods for the synthesis of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 17(11):1108–1129, November 1998.
34. C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn, Institut für Instrumentelle Mathematik, 1962. (technical report Schriften des IIM Nr. 3).
35. `petrify`: a tool for the synthesis of Petri nets and asynchronous controllers. http://www.lsi.upc.es/~jordic/petrify.
36. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998.
37. 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.
38. L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.
39. A. Valmari. Stubborn sets for reduced state space generation. In *Lecture Notes in Computer Science, Advances in Petri Nets 1990*, volume 483, pages 491–515. Springer Verlag, 1991.