An Approach to the Introduction of Formal Validation in an Asynchronous Circuit Design Flow

Dominique Borrione, Menouer Boubekeur, Emil Dumitrescu (VDS group)
Marc Renaudin, Jean-Baptiste Rigaud, Antoine Sirianni (CIS group)
TIMA Laboratory, Grenoble, France
E-mail for all authors: Surname.Name@imag.fr

Abstract

This paper discusses the integration of model-checking inside a design flow for Quasi-Delay Insensitive circuits. Both the formal validation of an asynchronous behavioral specification and the formal verification of the synchronous synthesis result are considered. The method follows several steps: formal model extraction, model simplification, environment modeling, writing temporal properties and proof. The approach is illustrated on a small, yet characteristic, asynchronous selection circuit.

1. Introduction

Design flows are well established for synchronous designs, and supported with efficient synthesis, simulation and verification software, starting from the Register Transfer level. Asynchronous designs have long been neglected, due to their higher bit cost, and tuning difficulty. With the advent of large transistor budgets, physically distant circuit modules communicate through long interconnections, and the global synchrony hypothesis is no longer realistic. The tendency for systems on a chip design is the reuse of existing components, with a distributed control and asynchronous interfaces. One of the objectives of our on-going research is to investigate the verification of mixed synchronous and asynchronous circuits, starting from a high-level specification.

Typically, asynchronous circuits are specified at logic level, using a CCS or a CSP-like formalism [Mar90, Ber93]. In our work, we write the initial behavioral description in an extension of the CHP language (from Caltech). Following the works of A. Martin [MLM97, MLM99], most of the initial effort at TIMA has been devoted to identifying the conditions required on the initial description, and correctness preserving transformations, in order to synthesize asynchronous circuits, starting from a high-level specification.

In this paper, after discussing the design flow, we describe the main CHP communication and synchronization primitives, their Petri Net model, their “pseudo clocked” VHDL translation, and the verifications that could be performed on the model, before and after synthesis. We then propose a set of transformations to produce a more compact and efficient coding of the state machine. As a running example, we use a simple, yet characteristic, selector module, and show examples of safety and liveness properties that have been verified.
2. The TAST Design Flow

TAST (Tima Asynchronous Synthesis Tools) [DR02] is an open design framework devoted to asynchronous circuits. It mainly consists in three parts: a compiler, a synthesizer and a simulation-model generator (Fig. 1). TAST offers the capability of targeting several outputs from a high level, CSP-like, description language called CHP (Communicating Hardware Processes).

The compiler translates CHP programs into Petri Nets (PN) associated to Data Flow Graphs (DFG). Such a model has been used for years to describe synchronous circuits and systems, but is particularly adequate for asynchronous digital circuits and systems design.

The synthesizer generates asynchronous circuits from the PN representation of the CHP programs. It is based on a Data Transfer Level specification. It provides a set of rules to guarantee that PN-DFG graphs are synthesizable into one of these two kinds of asynchronous circuits: Micropipeline [Sut89] and Quasi Delay Insensitive [Mar90]. In the following, only QDI circuits are discussed.

Behavioral VHDL models of the CHP specification are generated to verify them by simulation (simulation model generator).

Figure 2: Formal verification flow

The introduction of a formal verification flow in TAST starts from the PN–DFG format. (Fig. 2). First, one chooses an appropriate communication protocol, and all communication primitives are expanded according to this protocol. A state encoding is associated to the resulting Petri Net, based on its global place marking. Then a Finite State Machine (FSM) interpretation of the Petri Net is constructed and implemented as a VHDL behavioral model, which can be directly fed to an industrial model-checking tool, accepting a standard VHDL entry. The formal verification task consists in modeling the environment of the description, and writing a set of temporal properties that need to be satisfied.

On the other hand, the asynchronous synthesis tool produces a VHDL gate level net-list, which can also be fed to a model-checker. Thus, the compliance of the synthesized model with respect to its specification can be checked, by proving the same set of temporal properties on the specification and the synthesis result.

3. From CHP to Petri Nets

There is no agreement today on a specification language that provides all the facilities to model and synthesize asynchronous circuits. However, CSP-like languages are widely used. Caltech University has proposed CHP [Mar93], Philips has defined Tangram [BKR91][VB93], the University of Utah has developed a tool based on Occam [BS89], and Manchester has defined BALSA [BE97]. All these languages use the basic concept of CSP : concurrent processes communicating with channels [Hoa78].

We have developed a proprietary high level description language derived from CHP with specific features to cope with communication protocols, data encoding, arbitrary precision arithmetic, non-deterministic data flow, hierarchy, project management and, traceability. All these features make our modified CHP a very practical system description language to develop with[DR02].

3.1. CHP syntax basics

The CHP language is now briefly introduced, to provide the minimum understanding of the examples in this paper.

Literals

Integers constants are noted in fixed precision, as a sequence of dot separated digits, with a base indication. Example :

"1.9.7.9"[10] is in base 10 number 1979.
"1.2.3"[4] = 1*4^2 + 2*4^1 + 3*4^0 = 27.

For usual bases such as 10 and 2, standard notations are also supported for integers and binary numbers.

Data types

MR[B] : Multi-Rail in base B
This type represents a number between 0 and (B–1), coded with the “1-of-n” delay insensitive code.

Example :

VARIABLE b : MR [B] ;
 b := "x" [B] ; -- assign, with 0 <= x < B
MR[B][L] defines an array of L elements of type MR[B].

Based on these basic unsigned types, other types are defined to make the designers’ life easier, e.g.

DR: Dual Rail – equivalent to MR[2]
BIT, BOOLEAN: Binary – equivalent to MR[2]

All the types previously presented have a signed equivalent.

Operators

CHP provides the most common operators for comparison, arithmetic and logical operations. Language-specific operators describe sequential execution-“;” and parallel execution-“,”. Communication actions are described using the “!”(send), “?” (receive), “#” (probe) primitives.

Control structures

Deterministic selection. It waits for a unique guard to evaluate to true; it then executes the associated bloc and terminates the selection.

\[
\begin{cases}
\text{guard1 } => \text{ bloc1} \\
\text{guard2 } => \text{ bloc2} \\
\text{...}
\end{cases}
\]
Deterministic loop. While a unique guard is true, it executes the corresponding bloc. It terminates when none of the guard is true.

```
* [ guard1 => bloc1
  @ guard2 => bloc2
  @ ...
]
```

CHP also provides non-deterministic selection and loop structures. They are not used within this paper.

Program structure
A CHP COMPONENT is made of its communication interface, followed by a declaration part and its body. The communication interface is a directed port list, similar to VHDL. The declaration part declares local objects like channels and constants. The body is made of concurrent processes or component instances. They can all communicate with each other, and also with the component ports. Point to point communication only is allowed.

```
COMPONENT component_name
  PORT (port_list)
  {declaration part}
  Begin
  component body
  End component_name;
```

A process is made of a port list, a declaration part and a body.

```
PROCESS process-name
  PORT (port_list)
  {declaration part}
  [ instruction_list ]
```

Example
The program of Fig. 3 specifies a selector. Because there is only one process, the component and the process port lists are identical. Channel \( C \) is read in the local variable \( ctrl \) which is tested using a deterministic choice structure. The value read from channel \( E \) is propagated to channel \( S1 \) if \( ctrl \) is 0, to channel \( S2 \) if \( ctrl \) is 1, and to both \( S1 \) and \( S2 \) in parallel if \( ctrl \) is 3.

```
COMPONENT Selector
  PORT ( E: in DR;  C: in MR[3][1];  S1, S2 : out DR )
  Begin
  PROCESS main
    PORT (C: in MR[3][1];  E: in DR; S1, S2 : out DR)
    Variable x : DR;
    Variable ctrl : MR[3][1];
    [ *[ C ? ctrl ; [ ctrl = "0"[3] ] => E ? x; S1 ! x
      @ ctrl = "1"[3] ] => E ? x; S2 ! x
      @ ctrl = "2"[3] ] => E ? x; S1 ! x, S2 ! x
    ]
  End Selector
```

Figure 3: CHP code of a Selector

3.2. Petri Net generation

The language control structures, the sequential and parallel operators have a corresponding Petri Net model, shown on Fig. 4 to 7. Other instructions and expressions are represented using Data Flow Graphs: instructions are associated to the PN places, whereas guards are associated to transitions.

For the sequential operator Figure 6a, statement \( C2 \) is executed after the completion of instruction \( C1 \). Conversely, in Figure 6b, statements \( C1 \) and \( C2 \) are executed concurrently.

```
Figure 4: Selection operator : [Gi => Ci]
```

```
Figure 5: Repetition operator : *[Gi => Ci]
```

```
Figure 6: a : Sequential operator : C1 ; C2
b : Parallel operator : C1 , C2
```

Example
Fig. 7 gives the Petri Net obtained from the CHP program of Fig. 3. Place \( P0 \) is the first place of the repetition. This repetition has a unique implicit guarded command which is \([True => C? ctrl…]\). Hence, when the command execution completes, the execution has to restart from place \( P0 \). Moreover, place \( Pi \) is the initial place at the beginning of the execution: \( Pi \) has to be initially marked.

The first statement to execute is “\( C ? ctrl \)”. This instruction is represented by a DFG and is associated to place \( P1 \). It is followed by a sequential operator, modeled with transition...
Then the next instruction is a selection starting from place P9. Within the third guarded command note that the parallel operator is represented by places P3, P4 and transitions T3 and T5.

![Figure 7: Petri Net of the Selector](image)

### 4. From Petri Nets to gates

CHP programs have to be DTL-compliant (Data Transfer level) to ensure a quasi delay insensitive synthesis of the Petri Nets [DF02]. In order to derive a gate implementation from the Petri Net, channel and variable encoding as well as communication protocols have to be precisely defined.

#### 4.1. Data encoding

As discussed in section 3.1, all data are declared and built using the basic MR[B] type. For quasi delay insensitive hardware, a digit of type MR[B] is physically implemented with B rails, each wire or rail carrying out one of the B possible values between 0 and B-1 (one hot coding). For instance channels use the same encoding convention. In addition, an acknowledge signal is added to support handshaking communication protocols. Fig. 8 depicts the hardware representation of variable "ctrl" and channel E of the CHP program of Fig. 3.

![Figure 8: hardware implementation of object types](image)

#### 4.2. Communication protocols

The 4-phase handshaking protocol is chosen for implementing inter-process communications. Fig. 9 illustrates a data transfer along channel E (from CHP program of Fig. 3).

![Figure 9: transfer of value 0 on channel E](image)

Initially, rail0 and rail1 are both zero and the acknowledge signal "ack" is one. In the first phase, the event on "rail0" indicates that a zero is ready in channel "E". In response to this transfer request, phase 2 acknowledges the data by falling down signal "ack". Phase 3 and 4 are necessary to return the signals back to their initial values. A read action from channel "E" using this protocol can be formally described using Petri Nets as shown in §5.2. This representation is called a "handshaking expansion" [Mar90].

#### 4.3. Synthesis

Generating a gate net-list from the Petri Nets is a difficult problem that is beyond the scope of this paper. The first step of the synthesis process consists in expanding all the communication actions associated to the places of the Petri Net. This is done according to the chosen protocol as illustrated in §5.2. Then, all the instructions and guards associated to places and transitions, and represented by DFG constructs, are translated into gates. Finally, the gate net-list is generated from the expanded Petri Net and the synthesized DFG’s. The circuit obtained for the Selector is described in Fig. 10. Gates denoted "C" and "Cr" are Muller C elements without and with reset.

![Figure 10: Gate circuit of the Selector](image)

### 5. Verification

Verifying the correctness of delay-insensitive circuits involves two steps: validation of the initial specification and checking the correctness of the synthesized circuit.
In the first step we construct a finite-state machine description for the Petri net representation. After modeling the circuit environment, the resulting FSM is checked by applying model-checking tools.

The second step verifies the preservation of the logical properties of the asynchronous circuit after synthesis, with regard to the properties checked on the initial specification.

5.1 Petri nets as finite-state machines

The CHP specification is translated into a safe PN, formalized as a quadruple \( \Sigma = (S, T, F, M_0) \), where

(i) \( S \) and \( T \) are finite, disjoint, nonempty sets of places and transitions,

(ii) \( F \subseteq (S \times T) \cup (T \times S) \) is a flow relation,

(iii) \( M_0 : S \rightarrow \text{Bool} \) is the initial marking.

The data-flow graph part of the CHP translation is associated to the PN. Actions are attached to places and conditions are attached to transitions. Places with communication actions have to be expanded, according to the selected protocol. The guards associated with the PN transitions model the internal control flow and the synchronization of the communications. Each place is represented as a Boolean state-holding object and each marking as a min-term on these objects. This allows a direct translation of the PN as a FSM, where the marking represents the global state of the system. Firing the enabled transition \( t \) at a marking \( M \) produces a new marking \( M' \) constructed by setting all input places of \( t \) to 0 and all output places to 1.

5.2 Translation of the PN to a verifiable model

In order to benefit from existing industrial tools, and fit in the design flow, our prototype translates the PN representation into a register transfer level, behavioral VHDL model suited for formal verification (i.e. compliant to the 1076.6 standard). The simulation-oriented VHDL model generated in the TAST flow (Fig. 1) contains timing and attribute directives that are essential to follow the propagation of the signals, but make this model improper as an input to model checking tools. Ignoring or removing all delays would alter the behavior of the model. The trick consists in replacing each unit delay by a tick of a fictitious clock, thus making visible the “delta” delay of a purely behavioral, non-timed, state transition model.

The translation of the CHP COMPONENT interface is straightforward. All the internally declared variables and PN places are made VHDL signals, which guarantees that all state transitions take at least one step. The overall PN for a component is translated as a single process synchronized by the fictitious clock, and not as a set of guarded blocks as in [ABO98]. The full process is executed, and all active transitions are fired, at each step.

a) Global Algorithm

Input: Petri Net (list of places and transitions)
Output : VHDL program.
Translation of declaration part :
- declare each variable according to its type and its parameters (Base and Digit).
- For each element of the global Petri net
  IF (element is a place \( P \)) then
    associate a signal \( P \) to the place;
  IF (P is atomic)
    Print : \( \text{if } P \text{ then action}(P) \ldots \text{ end if} \);
    \(--\text{action}(P) : \text{Actions associated to place } P.\)
  else \( \text{The place expresses a communication.} \)
  \( \text{Put\_Com}() ; \)
ELSIF (element is a transition \( T \)) then
  Print :
    \( \text{if (And(\text{cond}(T), \text{places\_in}(T)))} \)
    \(--\text{cond}(T) : \text{Condition associated to transition } T.\)
    \( \text{places\_in}(T) \leq \text{false} ; \)
    \(--\text{Input places of } T.\)
    \( \text{places\_out}(T) \leq \text{true} ; \)
    \(--\text{Output places of } T.\)
End IF
End For

Remark. Put\_com() implements the communication expansion of Read? or Write! actions, according to the 4-phase handshake protocol, by insertion of the corresponding sub-Petri Net in the global Petri Net.

b) Translation of communications

The communication actions are implemented with the handshake protocol.

Translation of Read actions: \( (C\ ?\ ctr\_l) \)

\[ ((C="001" \text{ Or } C="010") \text{ or } C="100") \]
\[ \begin{align*}
  \text{C} &= \text{ctrl} ; \\
  \text{C\_ack} &= '0' ; \\
  \text{C} &= "000" \\
  \text{C\_ack} &= '1' \\
\end{align*} \]

**CHP declaration:**

\( C : \text{IN} \text{MR}[3][1]; \)
\( \text{variable} \text{ctrl: MR}[3][1]; \)

**Translation into VHDL:**

\( C : \text{IN} \text{bit\_vector}(2 \text{ downto } 0); \)
\( \text{C\_ack} : \text{OUT} \text{bit}; \text{--Acknowledgement} \)
\( \text{signal} \text{ctrl : bit\_vector}(2 \text{ downto } 0); \)

Translation of Write actions: \( (S!x) \)

\[ S \leq x ; \]
\[ S\_ack = '0' ; \]
\[ S \leq "00" ; \]
\[ S\_ack = '1' \]

**CHP declaration:**

\( S1, S2 : \text{OUT} \text{DR} \)
\( \text{variable} \text{x: DR}; \)

**Translation into VHDL:**

\( S : \text{OUT} \text{bit\_vector}(1 \text{ downto } 0); \)
\( S\_ack : \text{IN} \text{bit}; \text{--Acknowledgement} \)
\( \text{signal} \text{x : bit\_vector}(1 \text{ downto } 0); \)
entity EX11_Ent is
  port( C : in bit_vector(2 downto 0); E : in bit_vector(1 downto 0);
        S1 : out bit_vector(1 downto 0); S2 : out bit_vector(1 downto 0);
        C_a : out bit;  E_a : out bit;  S1_a : in bit;  S2_a : in bit;  clk, rst : in bit);
end EX11_Ent;

architecture EX11_a of EX11_Ent is
  signal EX11_MAIN_X : bit_vector(1 downto 0);
  signal EX11_MAIN_CTRL : bit_vector(2 downto 0);
  signal Pi, P0, P9, P7, P8, P1, P5, P6, P2, P3, P4, P0_1, P0_2, P7_1, P7_2,
      P8_1, P8_2, P5_1, P5_2, P6_1, P6_2, P2_1, P2_2, P3_1, P3_2, P4_1, P4_2 : boolean;
begin
  process(clk, rst)
  begin
    if (rst='0') then  -- initialization at Reset
      S1 <= "00"; S2 <= "00"; C_a <= '1'; E_a <= '1'; Pi <= true;
    elsif clk'event and clk='1' then  -- fictitious clock edge
      if P8 then S1 <= EX11_MAIN_X; end if;  -- start of Write action at place P8
      ... -- same for P6, P3, P4
      if P7_1 then EX11_MAIN_X <= E; E_a <= '0'; end if;  -- start of Read action at place P7
      ... -- same for P7
      if P7_2 then E_a <= '1'; end if;  -- acknowledge hand-shake at Expansion of P7
      if (P7_2) then P8 <= true; end if;  -- Transition T13
      if P8_1 then S1 <= "00"; end if;
      if ((EX11_MAIN_CTRL = "001") and P9) then P7 <= true; end if;  -- OR branch at P9
      if ((EX11_MAIN_CTRL = "010") and P9) then P5 <= true; end if;  -- OR branch at P9
      if ((EX11_MAIN_CTRL = "100") and P9) then P2 <= true; end if;  -- OR branch at P9
      if ((E="00") and P7_1) then P8_1 <= true; end if;  -- OR branch at P9
      if ((S1_a='0') and P8) then P8_1 <= true; end if;  -- OR branch at P9
      ... -- same for all other places and transitions
    end if;
  end process;
end EX11_a;

c) Application to the selector example

The above VHDL text gives excerpts from the automatic translation of the selector (Fig. 3), where many similar statements and line-feed characters have been deleted for space reasons. Comments were also manually added.

5.3 Verifying the asynchronous synthesis

Along the various synthesis steps, an erroneous procedure can produce an incorrect circuit description. It is thus useful to verify a synthesized circuit.

The circuit obtained after synthesis cannot be proven equivalent to its specification, because the synthesis process introduces optimizations and pipelines; thus the order of actions is not necessarily preserved. As a matter of fact, while the VHDL model for the specification is a single process per CHP process, the synthesized circuit is composed of many concurrent processes. We can only hope to prove that the essential safety properties, shown to hold for the initial specification, are preserved after synthesis.

entity MULLER2_R is
  port ( resetb, clk, A, B : in bit;  S : out bit);
end MULLER2_R;

architecture behaviour of MULLER2_R is
begin
  S<= s_s;
  s_s <= '0' after 1 ns when resetb = '0' else
    '1' after 1 ns when A='1' and B='1' else
    '0' after 1 ns when A='0' and B='0' else
    s_s;
end behaviour;

Figure 13: VHDL models of a Muller-C gate for simulation and for formal verification
Here again, we meet the same problem that was raised for the specification validation: the VHDL simulation model of the synthesized circuits does not comply to the standard verifiable VHDL subset. The asynchronous synthesis tool is based on a library that has to be adapted:
- Muller-C elements contain delays and transparent latches; we changed them into flip-flops, introducing a fictitious clock, as in section 5.2
- All types are transformed into Boolean and Bit.

Fig. 13 gives the transformation of one library component.

5.4 Environment modeling

In order to obtain a correct behavior, a set of environment assumptions must be associated to each communication channel, to guarantee its compliance to a four-phase communication protocol. Following the direction of the request signal attached to a channel, we distinguish between active channels (the system sends the request) and passive channels (the system receives the request). Thus, a dual behavior must be defined for each communication channel. For active channels, the dual behavior must receive the request and send an acknowledgement within a finite amount of time. Then, it waits until the request falls and deactivates the acknowledgement within a finite amount of time. The dual behavior of a passive channel must initiate a transaction, by rising the request signal at some arbitrary, but finite time. The remaining behavior is obtained from the active dual behavior, by exchanging the roles of the request and acknowledgement signals.

Figure 14: Modeling the environment

A dual behavior can be modeled as a non-deterministic description, which is plugged to each channel. A fairness condition must be associated to each dual instance, in order to express the fact that it always reacts within a finite delay. It is also possible to express a dual behavior as a set of assumptions written in temporal logic. Fig. 14 displays the composition of a system with the dual behaviors attached to each communication channel. Wire and is a pseudo-input signals used to model non-determinism.

5.5 Application to the selector example

In the case of the selector example, the environment has been described by a set of temporal formulas, and the verification was performed using FormalCheck[Cad01].

Input channel constraints

C_Env1
After P1 = True
Eventually (C = x"1" or C = x"2" or C = x"4") and P1 = True
Meaning: Each time place P1 is active, an incoming control request will eventually occur.

C_Env2
After C_ack = 0
Eventually C = x"0"
Meaning: Each time the request is acknowledged (C_ack = 0), it will eventually return to 0 (C = x"0").

Stable_C
After C = x"1" or C = x"2" or C = x"4"
Always C = stable
Unless C_ack = 0
Meaning: A request is stable until it is acknowledged

Output channel constraints

S1_Env1
After S1(1) = 1 or S1(0) = 1
Eventually S1_ack = 0
Meaning: After a request on S (S1(1)=1 or S1(0)=1), an acknowledgement will eventually be received (S1_ack = 1).

S1_Env2
After S1 = x"0"
Eventually S1_ack = 1
Meaning: After the write transaction on channel S is finished (S = x "0"), the acknowledgement will eventually be deactivated (S_ack = 1).

The constraints on ports E and S2 are similar to those on C and S1.

Some verified properties

The expressed properties correspond generally to the Petri net branches, i.e. the reachability of certain places or Transitions by following a given path. Or they are used to express input-output relationships, as in our example. The characteristic behavior of the selector is described by the three properties A1 to A3 below. These same properties are verified on the circuit specification and on the synthesized circuit.

Meaning of property A1: If place P01 is active and an incoming request C = x"1" arrives, then a write will eventually occur on S1.

A1) After P01 = True and C = x "1"
Eventually : S1(1) = 1 or S1(0) = 1

A2) After P01 = True and C = x "2"
Eventually : S2(1) = 1 or S2(0) = 1

A3) After : P01 = True and C = x "4"
Eventually : (S1(1) = 1 or S1(0) = 1) and (S2(1) = 1 or S2(0) = 1)
5.6 Handling combinational explosion

Applying symbolic model checking to a FSM interpretation of a CHP program, obviously the verification of large descriptions faces combinational explosion. Two main directions are explored in order to handle this problem. First, reduce the FSM state space. Currently, a FSM state is associated to each possible Petri Net place marking. This representation is excessively expensive, due to the state encoding which associates one state variable to each place. Second, a verification strategy is required, to exploit the characteristics (symmetry, control and data path, etc.) of particular classes of designs [RQ02], such as the asynchronous selector.

Refining the state machine model

A first improvement to our method consists in associating to the initial PN a FSM model with a more compact state encoding. For instance, states should be represented as values over an enumerated data type. Moreover, the FSM states and transitions should match the PN places and transitions. This correspondence is not always straightforward. "Split – join" (SJ) parallel execution constructions need special processing: generate one separate PN for each parallel branch, as well as the re-synchronization "glue" which marks the end of the parallel execution.

Fig. 15 illustrates this transformation. The parallel branches are executed as soon as place \( a \) is active and transition \( T1 \) is true. Each parallel branch makes a signal assignment (S1//S3) and waits for an external event to occur (Req1 = 1//Req2 = 1) before making a second assignment (S2//S4).

By successive applications of this transformation, we obtain a collection of concurrent PN's which are free of parallel execution statements. Since only one place at a time may hold value 1 in each concurrent PN, one unsigned signal per concurrent PN can hold the number of the active place. However, further simplifications are still possible on the resulting representation. Petri Nets allow the simultaneous use of two modeling levels in a natural way: sequences of local computations can be mixed with operations that imply waiting for an external event. Moreover, infinite execution paths that do not contain a wait for an external event are prohibited. Thus, the actual evolution of a PN is only triggered by external events. Any sequence of local computations performed between two external events can be considered as part of the same place, which only records their result. Hence, it is useful to collapse all sequences of places, which only implement local computations. Such sequences are characterized as follows: simple actions like assignments are associated to places; conditions are associated to transitions; one transition may fire as soon as its incoming place is active and its associated condition true. Fig. 16a, b and c present a few PN transformation patterns, which perform simplifications according to this criterion. We note them Sa, Sb and Sc.

The global simplification procedure consists in applying rules Sa and Sc on each appropriate PN construction. Before applying transformations SJ (Fig. 15) or Sb on a parallel construct, a preliminary analysis must determine whether its parallel branches implement any waiting on external events. This analysis performs a recursive coloring of each parallel PN construct:
- each transition is colored if it waits for an external event;
- each parallel structure is colored if its branches contain colored transitions or colored parallel structures.

Transformation SJ must be recursively applied on all colored constructs. Uncolored constructs are treated by
transformation $S_b$.

The FSM model obtained consists of several concurrent state machines that can be represented following a usual coding style. This simplification technique brings two major improvements:
- The resulting FSM state encoding relies on state enumeration. Our initial implementation associates one state variable to each PN place. For a PN containing $N$ places, $N$ state variables are needed. Concurrent FSM's state encoding would only need $\log_2 N$ state variables;
- Irrelevant PN places and transitions (if any) can be suppressed.

### Formal verification strategy

In the selector example of Fig. 7, once a channel is selected, its data is read and sent through an output channel. This sequence, call it a transaction, is repeated each time a new command is received. A transaction cannot influence the execution of a subsequent transaction. Hence, we may simply focus on writing correctness criteria for a single transaction. According to the value of the control word $ctrl$, the arbiter follows a separate execution path. Hence, each property can be split into three proof sub-goals, one for each execution scenario (corresponding to $ctrl = 0$, $1$, $2$), following two steps:
- abstract away the logic driving the variable $ctrl$, so that it becomes an artificial primary input;
- write a property for each execution scenario, by constraining $ctrl$ to the corresponding constant value.

If all sub-goals of a property pass, then we may consider that the initial property is true. This strategy seems adequate for this particular class of designs. It allows important simplifications in the FSM representation, as each sub-goal constrains $ctrl$ to a constant value.

### 6. Related Works

The verification of asynchronous circuits may follow two important directions, according to the asynchronous design approach: untimed (delay insensitive), or timed design [Be99]. It is well known that the verification of timed systems faces serious complexity problems. Based on the use of concurrent processes for specifying an asynchronous behavior, two main directions have been explored: language-based and graph-based asynchronous design.

Synthesis methods for language-based specifications directly translate a program into a circuit. In [Mar90], a series of semantic preserving transformations are applied. Graph-based specifications stand at a conceptual level lower than language based methods; they rely on Petri net or State Transition Graphs formalisms [Ro97].

Asynchronous verification methods have been developed following these design approaches. CIRCAL was experimented to evaluate correctness and performance properties on micro-pipelines [CM00]: both the system and its properties are modeled as parallel processes, whose composition is compared to the initial system model. If the proof succeeds, the system satisfies the property. This method has proved the correctness of two four-phase asynchronous micro-pipelines. [Cla99] presents two semantic models for the formal verification of reactive systems; they are based on simultaneous or interleaving approaches. It is argued that the interleaving model is more adequate for modeling and verifying asynchronous behaviors. The verification uses special partial order reduction techniques [BN01] in order to handle combinatorial explosion.

In [RC96] both the circuit specification and its environment assumptions are modeled using Petri nets. A state encoding is associated to this representation, which allows the application of BDD symbolic model checking techniques. [YG01] suggests the use of the LOTOS [EV89] specification language together with the CADP [CACP] toolbox for asynchronous verification using model checking.

All these approaches are subject to state space explosion. A number of techniques, such as hierarchical verification [RC95], modular verification, abstraction techniques [Zh01] and PN unfolding [McM92], already deal with this problem. As for RTL systems, the verification of asynchronous circuits heavily relies on the construction of a compact verifiable model as well as on using an adequate verification strategy.

### 7. Conclusion

We have implemented an asynchronous circuit design flow based on CHP and VHDL, which includes automatic synthesis, simulation and formal verification. A variety of small circuits (in the category of multiplexors, arbiters, and the like) have been formally verified, by model checking techniques, using pre-existing property checking tools that were initially not intended for asynchronous circuit verification. However, the current prototype FSM generator is not efficient enough to serve the verification of large circuits. Our on-going works concern the development of algorithms for generating more compact state machines, and for finding verification strategies adapted to the known characteristics of various circuit types, as described in the section devoted to the handling of combinatorial explosion. The implementation of these algorithms will be progressively added to the TAST environment for asynchronous circuit designs.

### 8. References


