IEE Proceedings-E Computers and Digital Techniques

Title:

VHDL Generation from Hierarchical Petri Net Specifications of Parallel Controllers

Authors:

- João Miguel Fernandes,
  Dep. Informática, Universidade do Minho, 4709 Braga Codex, PORTUGAL
  `miguel@di.uminho.pt`
- Marian Adamski,
  Dep. Electrónica Industrial, Universidade do Minho, 4800 Guimarães, PORTUGAL
  `madamski@eng.uminho.pt`
- Alberto José Proença,
  Dep. Informática, Universidade do Minho, 4709 Braga Codex, PORTUGAL
  `aproenca@ci.uminho.pt`
  Phone: 351-53-604144
  Fax: 351-53-612251

Abstract:

Parallel controllers can be best specified using a description with a formal support to validate structural and dynamic properties. Petri Nets (PN) can provide an adequate means to model and to animate parallel systems based on the control and data path approach, in a hierarchically structured way. A set of tools was developed to allow formal validation of parallel controllers, based on hierarchical PN-based specifications and to automatically generate RT-level VHDL code. An example of a VLSI chip design, the transputer link adaptor, shows the capabilities of this methodology and associated tools.

VHDL Generation from Hierarchical Petri Net Specifications of Parallel Controllers

João M. Fernandes* **Marian Adamski+** **Alberto J. Proença***

*Dep. Informática, Universidade do Minho
4709 Braga codex, Portugal

+Dep. Electrónica Industrial, Universidade do Minho
4800 Guimarães, Portugal

Abstract

Parallel controllers can be best specified using a description with a formal support to validate structural and dynamic properties. Petri Nets (PN) can provide an adequate means to model and to animate parallel systems based on the control and data path approach, in a hierarchically structured way. A set of tools was developed to allow formal validation of parallel controllers, based on hierarchical PN-based specifications and to automatically generate RT-level VHDL code. An example of a VLSI chip design, the transputer link adaptor, shows the capabilities of this methodology and associated tools.

1 Introduction

The use of Finite State Machines (FSMs) for the graphical specification of the control unit of a digital system is no longer adequate when the system presents parallel activities. Among the modelling paradigms, Petri Nets (PN) [1] allow an easy specification of cooperating subsystems and the use of formal validation methods, which are based on a mathematical theory. These methods should be applied prior to the implementation phase and in parallel with the simulation phase, to minimize the consequences of design errors.

Several types of PNs have been proposed to specify and model digital systems, either by imposing restrictions to a basic model, or by adding extensions to it. A PN-based controller can be best specified and modelled by a safe PN with guarded transitions and synchronous transitions firing; this PN should also support enabling and inhibitor arcs [2, 3].

The specification languages currently available in ECAD packages often lack appropriate support to clearly express concurrent and cooperating activities. A new software framework was developed to accept a PN-based controller specification as input, to validate the properties of the controller, to allow the PN model animation, and to generate the corresponding VHDL code. This code can feed standard ECAD packages for simulation and synthesis purposes.

This paper is organized as follows. The selected PN type and its relevant characteristics for digital systems are presented in Section 2. The advantages of merging PNs with VHDL are presented in Section 3. The ConPar language, which is used to specify parallel controllers based on PNs, is described in Section 4. In Section 5 the ECAD framework and the VHDL generation process are introduced. A detailed example closes this paper in Section 6 by clarifying the presented concepts.

2 Petri nets and digital systems

A design and implementation methodology must provide tools for system specification, modelling and implementation. System specification includes the description of the expected behaviour of the digital system. Modelling involves constructing a mathematical formalism embodying the specified system behaviour. This formalism can be manipulated and analysed to determine properties of the system which are not necessarily apparent from the initial system specification. The modelling formalism may also be adopted for the system specification, when there is a close correspondence between the system model and its specification.

PNs have shown to be a powerful tool to specify and model the behaviour of parallel systems, and in particular, parallel digital controllers. A detailed analysis of the model, based on a set of well established methods, allows the detection of a large number of design errors prior to the system implementation.

A marking of the PN can be regarded as a global state of the modelled system (node in the reachability graph) and a change of the marking corresponds to a state transition (edge in the reachability graph).

Safe PNs can be viewed as a natural extension to FSMs, providing an easy migration path from FSM to PN-based specifications. To realistically model any parallel controller with safe Place/Transition (P/T) nets [1], some well-known modifications are introduced:

- logic expressions are assigned to transitions (the guards);
- output signals are attached to places and transitions, to represent the controller actions;
- transitions firing are synchronized with the active edge of a (global) clock.

To achieve efficient testing and synthesis, the safe P/T net is treated as a Condition/Event (C/E) PN (in section 4). Moore type output signals are associated to places, while Mealy type output signals are related to transitions. The resulting PN type is a Synchronous Interpreted PN (SIPN). To implement an SIPN, all the enabled transitions at a given moment wait for a clock pulse and then all fire to produce a new marking. Fig. 1 (in [4]), presents an SIPN example, where `xi` are input signals and `yi` are output signals.

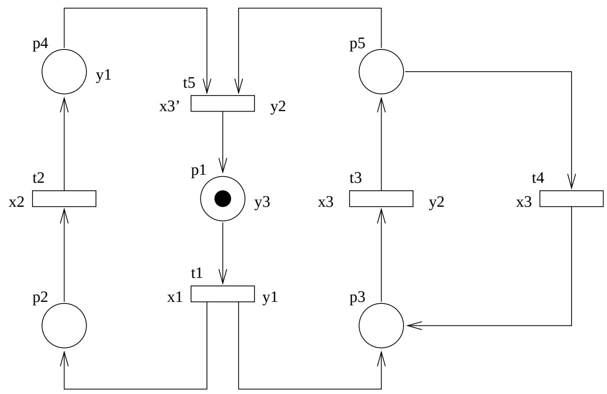


Figure 1: An SIPN specification of a controller.

Some quasi-independent sub-PNs may require priority and synchronization schemes to share resources. To support these requirements, enabling and inhibitor arcs are used.

A priority can be modelled with an inhibitor arc. An inhibitor arc, represented by a dashed line with a circle, connects a place to a transition and disables the transition when the place is marked. Consider a system in which two processes access a shared resource. A conflict arises when both processes apply for the resource. To solve this problem, an inhibitor arc is introduced as shown in Fig. 2.

A synchronization between two processes can be modelled by an enabling arc. An enabling arc, represented by a dashed line with an arrow, connects a place to a transition and enables the transition when the place is marked. Nevertheless, when the transition fires, no token is removed from the place connected to the transition through an enabling arc. Consider the system in Fig. 3 in which process B can continue only when a token is present in place `pA`.

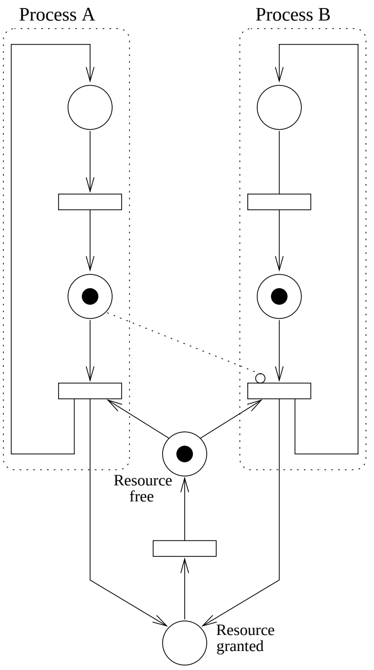


Figure 2: The use of an inhibitor arc and a shared place to model a priority.

Structured programming concepts are also useful to deal with the specification and modelling of complex or large controller systems. Hierarchical specification mechanisms can be introduced in macronodes (macroplaces or macrotransitions) [5] to allow the encapsulation of sub-PNs and their use as often as required. This approach decreases the size of the specification, improves its readability, and reduces the number of typing errors.

As an example of a hierarchical SIPN, consider the PN and the macroplace represented in Fig. 4(a) [6]. The resulting PN, after expanding the macroplace, is presented in Fig. 4(b).

A PN is said to be *live* if, from any global state, it can always enable all the transitions. This fact guarantees that the system is deadlock-free and that there is no dead code (transitions that are never enabled or places that are never marked). The latter ensures that there is no wasted silicon at the final implementation. To test liveness, a reachability graph of a PN is used: it describes all markings that are reachable from the initial marking. The reachability graph is usually built by generating all possible reachable markings, without considering its interpretation and synchronism.

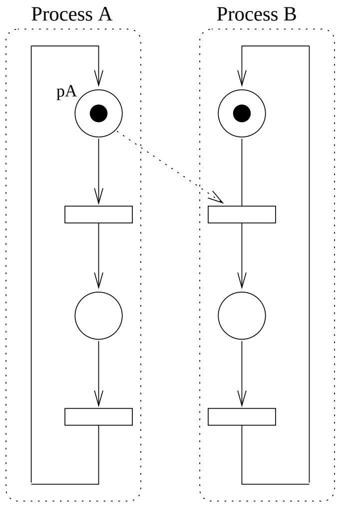


Figure 3: The use of an enabling arc to model a synchronization.

An undesirable occurrence on an SIPN occurs when two or more transitions attempt to simultaneously mark/unmark the same output/input place. This is illustrated in Fig. 5. The controller is not properly specified if it allows undesirable situations.

3 Petri nets and VHDL

PNs are well suited for modelling and formal analysis of complex discrete systems, while VHDL is a standard hardware description language which exploits concurrency in the specification of digital systems, allowing their simulation and synthesis. PNs and VHDL may complement each other and they may also provide a proof subsystem that accepts the same user interface descriptions for all design tasks [7, 8].

Modelling PNs in VHDL was already debated in 1990 [9]. A VHDL textual PN description of parallel controllers was proposed in [4], which describes a VHDL template with `ASSERT` statements to enable the syntatic and semantic correctness of the model to be tested. Experimental results, developed at Inmos in a practical design, achieved a 50% area reduction and a 40%

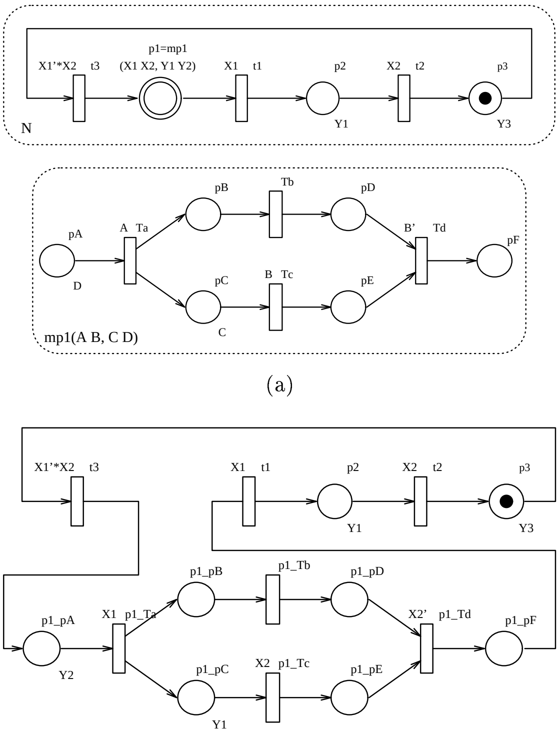


Figure 4: Hierarchy in SIPNs: (a) SIPN with a macroplace, (b) exploding the macroplace.

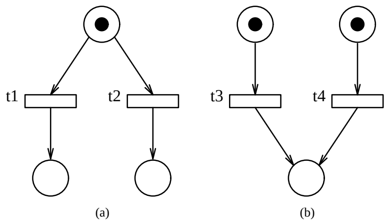


Figure 5: Undesirable occurrences on an SIPN: (a) conflict, (b) overflow.

speed improvement over the best FSM synthesis. Another straightforward VHDL model of PNs, adequate for automatic generation by a translation tool, was later presented [10]. The specification of PNs in terms of LSMs (Linked State Machines) and VHDL can be found, for example, in [11, 12].

Research on mapping coloured PNs into VHDL was also pursued at Linköping (Sweden) [13]. The commercial tools Design/CPN and MINT were used to map a controller into VLSI. As stated, the research concluded without any experimental validation of the synthesis efficiency, and the implemented examples were fairly trivial. Another technique to convert a textual PN design representation into a VHDL description was introduced by Peng [14] in CAMAD system. However, the PN was transformed into the classical FSM model, and the advantages of concurrency were lost.

The VHDL code generated by SystemSpecs [15] is mainly used for general modelling purposes and the efficiency of VLSI compilation results were not reported.

The PN modelling techniques for simulation were also introduced, among others, by Swaminathan *et al.* [16] and by Benders *et al.* [17], but related VLSI synthesis was not reported. The only efficient methodology, proven in industry, was introduced by Bolton and it was extended in this work. The approach followed here, for synchronous parallel controllers design, is closely related with the PARIS tool [3] and suggests a particular way to implement a safe PN specification of discrete systems in VHDL for later simulation and synthesis. It generates VHDL code and support hierarchical specifications.

To keep a very strict direct correspondence (isomorphism) between an initial PN specification, in a rule-based format, and VHDL statements, the authors propose the use of an intermediate language ConPar. The main goal is to preserve the one-to-one correspondence between simulation and synthesis specifications and to produce implementations with a high degree of concurrency.

Other methods were also proposed for a reverse transformation: from specifications defined in a VHDL subset to PNs for performance and reliability analysis. Some of the most recent developments related to modelling and analysis were reported, for example, in [18, 19, 20].

4 The ConPar description language

PNs can also be viewed as formal models for logic rule-based specifications [21, 22]. They make the straightforward link between algebraic numerical methods and the symbolic mathematical logic based methods of specification, optimisation, verification and synthesis. The rule-based form of specification can be considered as an alternative textual form of timing diagram description. The causality among signals is explicitly given in terms of local, relevant inputs, outputs and state changes.

Several researchers have independently proposed rule-based formalisms for the description and implementation of autonomous discrete concurrent systems, which are related with PNs. Transition rules are usually treated as production rules ('if-then' non-procedural statements) [21]. The rule-based description supported by means of logic deduction techniques (Gentzen natural logic calculus) was recently presented in the Logic Controller design context [23]. The syntax of the language was revised over the years. Some major syntax modifications have been made in PARIS system [2, 3, 24] and a new description format called the PN Specification Format (PNSF) was introduced.

The ConPar specification format (in appendix A) was developed as a bridge between the textual logic description of a PN and its VHDL model. It supports macroplaces and it is consistent with previously introduced rule-based specification languages. It also supports the translation of a hierarchical symbolic representation of PNs (with macroplaces and macrotransitions) directly into VHDL format.

Restricted interpreted PNs are used in this work as the basic specification formalism. They are translated into rule-based specifications, which are composed of discrete local state symbols, input and output signal symbols of the controller. Discrete state transition rules describe local state changes, influenced by the external environment. The rule-based description does not explicitly describe sequencing, but sequences of operations could be easily derived by ordering the transition rules.

In ConPar notation, a transition is described as a conditional rule:

```
<label> : <PreConditions> |- <PostConditions> ;
```

The precondition and postcondition are respectively formed from input and output place symbols. When the preconditions of a rule are satisfied (hold), the postconditions are made true (they will hold). Logical conjunction of all related discrete states is assumed when the precondition contains more than one discrete state symbol.

For example, the transition `t1` in Fig. 1 has input place `p1`, output places `p2` and `p3`, it is guarded by input `x1` and the output signal `y1` is activated when the transition is enabled. In ConPar notation, this transition is described as follows:

```
t1: p1 * x1 |- p2 * p3 * y1;
```

To obtain an efficient implementation, the PN may be directly mapped into boolean equations without explicit enumeration of all possible global states and global state changes [22]. The specification is given in terms of the local states changes (local transitions) and one-hot code state assignment is used [4, 25]. In Peng's approach [14], the VHDL code is not directly related to the original PN specification, which causes some implementation inefficiency. As mentioned earlier, the PN is transformed into a FSM (the FSM is built in the same way as the reachability graph) and then translated into VHDL using a `CASE` statement inside a `PROCESS`.

Consider again transition `t1` in Fig. 1. Its description in VHDL is presented below and can be read as: "transition `t1` will be enabled when input signal `x1` is asserted, place `p1` is marked and places `p2` and `p3` are not marked".

```
t1 <= x1 AND p1 AND NOT p2 AND NOT p3;
```

The preconditions of a given transition are directly mapped into a VHDL boolean conjunction. The postconditions are distributed among the VHDL expressions Npi ($i = 1 \ldots 5$), which are VHDL signals that model inputs to flip-flops. For transition t1, the relevant parts of the expressions are as follows:

```
Np1 <= ... OR (p1 AND NOT t1);
Np2 <= t1 OR ...;
Np3 <= ... OR t1 OR ...;
```

The assignment for Np1 describes place p1 holding the token, when transition t1 is not enabled. The remaining assignments represent places p2 and p3 when they get new tokens, when transition t1 fires.

For each output signal, a concurrent signal assignment is used, which describes the nodes where the signal is activated. For output signal y1 in Fig. 1, the assignment is written as follows:

```
y1 <= ... OR t1;
```

5 ECAD framework

A new software framework [6, 26] was developed to feed any ECAD package that accepts VHDL as input. This framework is appropriate for small controller systems specified at the RT-level. The hierarchical PN specification is directly and efficiently mapped to boolean equations. This approach simplifies the VHDL code debbuging, since there is a direct correspondence between the original PN, the ConPar description and the produced VHDL code.

The complete framework, illustrated in Fig. 6, provides editors for the SIPN specification, analyses the basic properties of the SIPN, and compiles the specification into VHDL.

The text editor uses the ConPar language notation. As an illustration, the ConPar description for the SIPN in Fig. 1 is listed in Fig. 7(a). A graphic SIPN editor, which is currently under development, can offer a better user friendly interface, hiding the ConPar language formalities. The SIPN can be hierarchically specified, using expanded macroplaces.

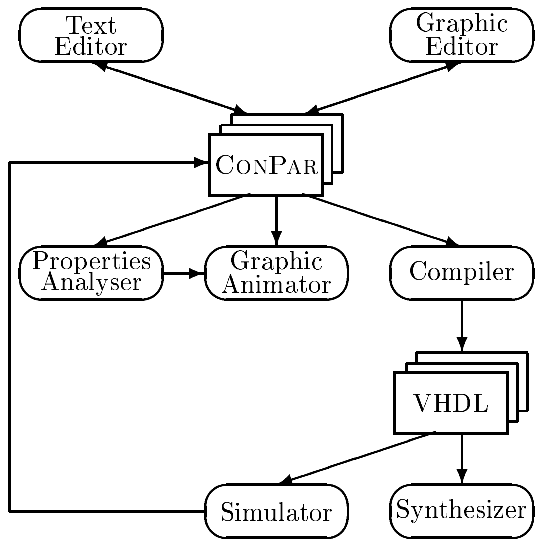


Figure 6: The complete framework.

It is possible to adapt already available commercial editors for PNs, such as, the editor supplied by the SystemSpecs tools. In general, those editors are too expensive and too general for the restricted application area considered in this project.

The properties analyser verifies if the input specifications are live and conflict-free, issuing a message to the user, whenever a problem occurs (deadlock or conflict). Since it is not appropriate to mark a place if it is already marked, that situation is also detected, clearly located and an adequate message is sent to the user interface.

The algorithm used in the properties analyser to verify the liveness builds the reachability graph for an SIPN (a restricted reachability graph) as defined, for example, in [3, 14] and uses the PN structure. The liveness of the SIPN is partially analysed through the following tests:

1. to check if there are source or sink places;
2. to ensure that each marking has, at least, one successor marking;
3. to verify that each transition is reached at least once, connecting two consecutive markings in the reachability graph.

The analysis procedures, on the current framework prototype, are only applied at the outmost abstract level: when the PN is specified in a hierarchical way (see figure 4 for example), only the main PN is tested for validation. To overcome this limitation, the methods introduced by Valette [5] and Suzuki *et al.* [27] will be considered. These methods show that it is possible to separatelly analyse the main PN and the macronodes, and to extrapolate the analysis results for the global PN.

The graphic SIPN animator allows the user to execute the PN model that gives structural support to the controller being implemented. The current animator implementation [28] is an independent graphic application, built on top of OBJECTWORKS/SMALLTALK, that allows the user to directly interact with a running PN. The interface is used to display the information that represents the current state of the SIPN, to send the input signals and to activate the transitions.

The style of description for the code generated by the compiler is similar to the template type presented by Bolton *et al.* [4]. Some modifications were introduced to support the adopted PN model. To make the implementation more efficient, negated output places are introduced in the transition expressions. This also simplifies the SIPN testing and forces the system to be stopped if the one-hot mapping is partially destroyed during the evaluation of real hardware.

The current compiler version provides two code generation alternatives, according to the ECAD package tool being used. To infer the initial and next markings, the user can select either a `BLOCK` statement or a `PROCESS` statement to be included into the generated VHDL file. In both cases, the produced VHDL code is at the RT-level.

As an example, the generated VHDL code, for the SIPN in Fig. 1, is listed in Fig. 7(b), which includes a `PROCESS` statement. The `ENTITY` and `PROCESS` used to infer the state register are identical to those used for an FSM description in VHDL [10, 11, 14]. However, whilst an FSM usually includes a `CASE` statement to infer the combinational logic, the SIPN uses concurrent signal assignments, including a list of transitions, next place markings and control outputs.

To simplify fast validation by VHDL simulation, `ASSERT` statements are automatically included in the produced code [4]. These statements check for potential conflicts on transitions and test for possible deadlock situations. For the latter, a warning is issued since the controller may be just waiting for transition guards to become true.

For hierarchical SIPNs, the current compiler version generates flat (one level) VHDL code. In Section 6, an example is considered using a hierarchical SIPN and the generation of VHDL code with a `BLOCK` statement is presented.

The compiler was built with the facilities provided by the Compiler Tool Box (CTB) [29]. This tool automatically generates routines that accomplish the lexical, syntactic and semantic analysis, from several specifications. CTB allows efficient compilers to be developed, by providing tools that cover almost all compilation tasks.

The generated programs — from the CTB and manually — are ANSI C, allowing easy modification and compiler portability to different platforms.

6 Design example: the transputer link adaptor

To clarify the concepts presented in the previous sections, a detailed example is presented: the transputer link adaptor [30, 2], which interfaces a transputer network with its host bus system. A transputer network uses serial links to perform nearest-neighbour communication. To enable full duplex data flow, two wires are used for each link, and data transmitted along the outgoing wire is synchronized by acknowledgments from the receiving transputer on the incoming wire. Thus data, and acknowledgments for received data, are interleaved on each wire.

```
.clock RELOGIO
.input x1 x2 x3
.output y1 y2 y3

.part controller
.place p1 p2 p3 p4 p5
.transition t1 t2 t3 t4 t5
.net
t1: p1 * x1 |- p2 * p3 * y1;
t2: p2 * x2 |- p4 ;
t3: p3 * x3 |- p5 * y2;
t4: p5 * x3 |- p3;
t5: p4 * p5 * !x3 |- p1 * y2;
.MooreOutput
p1 |- y3;
p4 |- y1;
.marking p1
.e
```

(a)

```
ENTITY controller IS
  PORT (reset, x1, x2, x3, clock : IN BIT;
        y1, y2, y3 : OUT BIT);
END controller;

ARCHITECTURE dataflow OF controller IS
-- Place Signals
  SIGNAL p1, Np1 : BIT;
  SIGNAL p2, Np2 : BIT;
  SIGNAL p3, Np3 : BIT;
  SIGNAL p4, Np4 : BIT;
  SIGNAL p5, Np5 : BIT;
-- Transition Signals
  SIGNAL t1 : BIT;
  SIGNAL t2 : BIT;
  SIGNAL t3 : BIT;
  SIGNAL t4 : BIT;
  SIGNAL t5 : BIT;
BEGIN
  PROCESS BEGIN
    WAIT UNTIL clock'EVENT and clock='1';
    IF reset='0' THEN
      p1 <= Np1;
      p2 <= Np2;
      p3 <= Np3;
      p4 <= Np4;
      p5 <= Np5;
    ELSE
      p1 <= '1';
      p2 <= '0';
      p3 <= '0';
      p4 <= '0';
      p5 <= '0';
    END IF;
  END PROCESS;
-- Dataflow description for transitions
  t1 <= NOT p2 AND NOT p3 AND x1 AND p1;
  t2 <= NOT p4 AND x2 AND p2;
  t3 <= NOT p5 AND x3 AND p3;
  t4 <= NOT p3 AND x3 AND p5;
  t5 <= NOT p1 AND p5 AND p4 AND (NOT x3);
-- Dataflow description for next place markings
  Np1 <= t5 OR (p1 AND NOT t1);
  Np2 <= t1 OR (p2 AND NOT t2);
  Np3 <= t4 OR t1 OR (p3 AND NOT t3);
  Np4 <= t2 OR (p4 AND NOT t5);
  Np5 <= t3 OR (p5 AND NOT t5 AND NOT t4);
-- Output Signals Equations
  y1 <= p4 OR t1;
  y2 <= t5 OR t3;
  y3 <= p1;
---- ASSERT Commands
-- Transitions in conflict
  ASSERT NOT (t4 AND t1)
    REPORT "output place p3 overflows (transitions t4,t1)"
    SEVERITY ERROR;
  ASSERT NOT (t5 AND t4)
    REPORT "t5 and t4 are in conflict (input place p5)"
    SEVERITY ERROR;
-- No Enabled Transitions
  ASSERT NOT (t1='0' AND t2='0' AND t3='0' AND t4='0' AND t5='0')
    REPORT "Petri Net may be deadlocked"
    SEVERITY WARNING;
END dataflow;
```

(b)

Figure 7: (a) ConPar description, (b) generated VHDL code for a linear SIPN.

A data packet is transmitted as two start bits, followed by the data byte and the stop bit. The transmitter then waits for an acknowledgment packet, consisting of a start bit followed by a stop bit. To enable continuous data transfer, the receiver can transmit an acknowledgment packet as soon as an incoming data header is detected (provided it is not also transmitting a data packet at that time).

The communication between the transputer network and an external bus-based system requires serial-to-parallel and parallel-to-serial conversions to be performed at the network. The link adaptor shown in Fig. 8 provides a full duplex interface between a transputer link and two undirectional 8-bit buses. Data on the input bus (`I0-I7`) is multiplexed onto `LinkOut` whilst data on `LinkIn` is latched, via an 8-bit register, onto the output bus (`Q0-Q7`). A complete handshake protocol (`IValid, IAck, QValid, QAck`) controls data transfer to and from the buses.

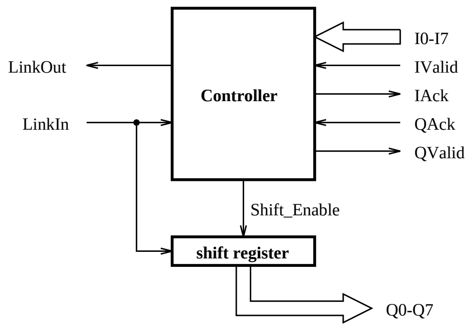


Figure 8: The link adaptor.

Parallel-to-serial conversion is described by the diagram in Fig. 9(a). The bus driver deposits a data byte on signals `I0-I7` and raises `IValid`. The link adaptor packages the data, multiplexes it onto `LinkOut`, and looks for an acknowledgment packet on `LinkIn`. When this acknowledgment is received, the link adaptor relays it to the bus by raising `IAck`. The bus driver replies by lowering `IValid` and waits for the link adaptor to lower `IAck`.

Serial-to-parallel conversion is described by the diagram in Fig. 9(b). The transputer sends a data packet along `LinkIn`. On detecting the incoming data header, the link adaptor discards the packaging, latches the data byte on `Q0-Q7`, and asserts `QValid`. When `QAck` is raised in reply, the link adaptor relays it to the transputer as an acknowledgment packet on `LinkOut`. Then, the link adaptor lowers `QValid` and waits for the receiver to lower `QAck`.

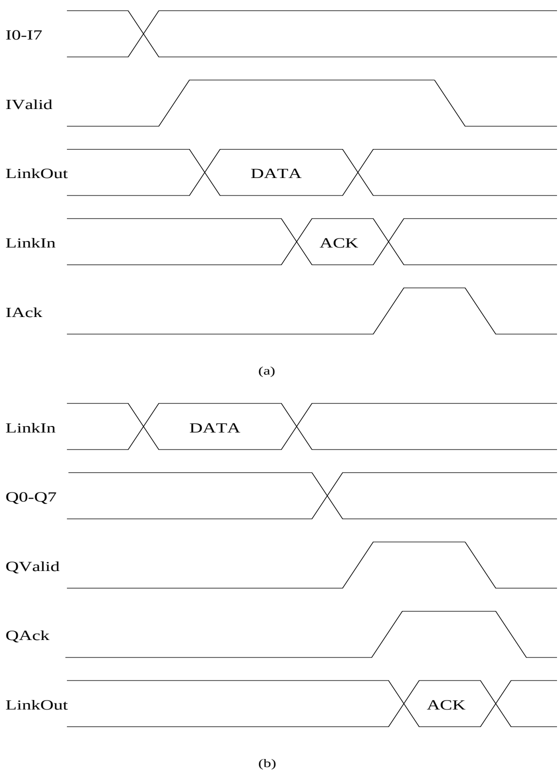


Figure 9: Timing diagrams: (a) parallel-to-serial conversion, (b) serial-to-parallel conversion.

The system controller of the link adaptor can be specified by the SIPN in Fig. 10(a). The design, a revision of an earlier prototype [2], is a concurrent Mealy machine comprising 29 places and 35 transitions. Two macroplaces —Fig. 10(b) and (c)— were introduced, which reduces the size of the specification and improves its readability.

The PN can be roughly divided into four partially overlapped parts:

- places `p1`, `p2`, and macroplace `SerPar` perform serial-to-parallel conversion and `LinkIn` acknowledge packet detection.
- places `p28`, `p11`, `p12`, and macroplace

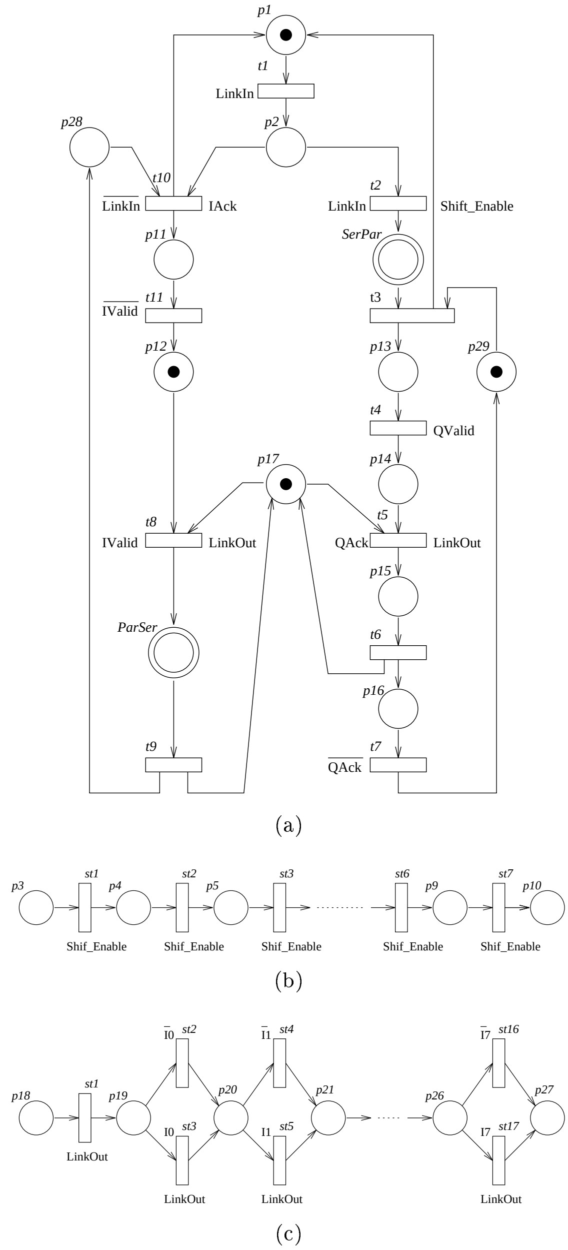


Figure 10: SIPN for the link adaptor controller: (a) the main PN, (b) the serial-to-parallel conversion macroplace, (c) the parallel-to-serial conversion macroplace.

ParSer perform parallel-to-serial conversion and LinkOut acknowledge packet sending.

- places p13-p16, p29 represent the second part of serial-to-parallel conversion: generating QValid output status signal, and waiting for acknowledge signal on the input QAck.

- places p17 (added for shared resources control on the output LinkOut), p15 and macroplace ParSer constitute the last concurrent part.

The textual specification of the system controller in ConPar notation is listed in Fig. 11.

Using this notation to feed the properties analyser module, a potential conflict situation, similar to the one presented in Fig. 5(a), is detected: transitions t5 and t8 share the input place p17. Notice that those transitions are guarded by predicates QAck and IValid, respectively. To solve the possible conflict detected by the properties analyser, an inhibitor arc from p14 to the transition t8 may be introduced, as depicted in Fig. 12.

A file containing the reachability graph can be created by the properties analyser module. A part of the produced graph related to the initial marking $\{p29, p17, p12, p1\}$ follows:

```
...
p29 p17 p12 p1   : t8 t1   -> parser p29 p2
                 : t8      -> parser p29 p1
                 : t1      -> p29 p17 p12 p2
...
```

With the introduced modification (the inhibitor arc), the analysis phase did not report any severe error and a data flow VHDL file is produced by the compiler module (Fig. 13). A compilation option was selected to direct the compiler to create a BLOCK statement.

The ASSERT statements, automatically generated by the compiler tool, help the user in the system simulation. Those statements roughly detect transitions in conflict and deadlock situ-

```
.CLOCK clock
.INPUT I0 I1 I2 I3 I4 I5 I6 I7 LinkIn QACK IValid
.OUTPUT LinkOut IAck QValid ShiftEnable
<*************************************************>
.MACROPLACE PS(in0 in1 in2 in3 in4 in5 in6 in7, out)
.INTERFACE p18, p27
.PLACE p19 p20 p21 p22 p23 p24 p25 p26
.TRANSITION st1 st2 st3 st4 st5 st6 st7 st8 st9
        st10 st11 st12 st13 st14 st15 st16 st17
.NET
   st1: p18         |- p19 * out;
   st2: p19 * !in0  |- p20;
   st3: p19 *  in0  |- p20 * out;
   st4: p20 * !in1  |- p21;
   st5: p20 *  in1  |- p21 * out;
   st6: p21 * !in2  |- p22;
   st7: p21 *  in2  |- p22 * out;
   st8: p22 * !in3  |- p23;
   st9: p22 *  in3  |- p23 * out;
  st10: p23 * !in4  |- p24;
  st11: p23 *  in4  |- p24 * out;
  st12: p24 * !in5  |- p25;
  st13: p24 *  in5  |- p25 * out;
  st14: p25 * !in6  |- p26;
  st15: p25 *  in6  |- p26 * out;
  st16: p26 * !in7  |- p27;
  st17: p26 *  in7  |- p27 * out;
<*************************************************>
.MACROPLACE SP(in, out)
.INTERFACE p3, p10
.PLACE p4 p5 p6 p7 p8 p9
.TRANSITION st1 st2 st3 st4 st5 st6 st7
.NET
   st1: p3 |-  p4 * out;
   st2: p4 |-  p5 * out;
   st3: p5 |-  p6 * out;
   st4: p6 |-  p7 * out;
   st5: p7 |-  p8 * out;
   st6: p8 |-  p9 * out;
   st7: p9 |- p10 * out;
<*************************************************>
.PART macronet
.PLACE p1 p2 p11 p12 p13 p14 p15 p16 p17 p28 p29
        SerPar=SP(I0,ShiftEnable)
        ParSer=PS(I0 I1 I2 I3 I4 I5 I6 I7, LinkOut)
.TRANSITION t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11
.NET
   t1: p1 * LinkIn          |- p2;
   t2: p2 * LinkIn          |- SerPar * ShiftEnable;
   t3: SerPar * p29         |- p1 * p13;
   t4: p13                  |- p14 * QValid;
   t5: p14 * p17 * QACK     |- p15 * LinkOut;
   t6: p15                  |- p16 * p17;
   t7: p16 * !QACK          |- p29;
   t8: p12 * p17 * IValid   |- ParSer * LinkOut;
   t9: ParSer               |- p28 * p17;
  t10: p2 * p28 * !LinkIn   |- p1 * p11 * IAck;
  t11: p11 * !IValid        |- p12;
.MARKING
   p1 p12 p17 p29
<*************************************************>
.E
```

Figure 11: ConPar code for the link adaptor controller.

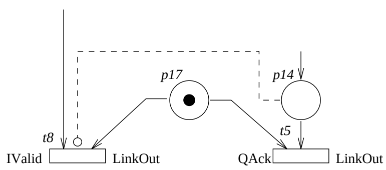


Figure 12: Inhibitor arc to solve a conflict situation.

```
ENTITY controller IS
  PORT (reset, i0, i1, ... i7, linkin, qack, ivalid, clock : IN BIT;
        linkout, iack, qvalid, shiftenable : OUT BIT);
END controller;

ARCHITECTURE dataflow OF controller IS
-- Place Signals
  SIGNAL p1  : REG_BIT REGISTER;
  SIGNAL Np1 : BIT;
  ...
  SIGNAL p29  : REG_BIT REGISTER;
  SIGNAL Np29 : BIT;
  SIGNAL serpar_p3  : REG_BIT REGISTER;
  SIGNAL Nserpar_p3 : BIT;
  ...
  SIGNAL serpar_p10  : REG_BIT REGISTER;
  SIGNAL Nserpar_p10 : BIT;
  SIGNAL parser_p18  : REG_BIT REGISTER;
  SIGNAL Nparser_p18 : BIT;
  ...
  SIGNAL parser_p27  : REG_BIT REGISTER;
  SIGNAL Nparser_p27 : BIT;
-- Transition Signals
  SIGNAL t1 : BIT;
  ...
  SIGNAL t10 : BIT;
  SIGNAL serpar_st7 : BIT;
  ...
  SIGNAL serpar_st1 : BIT;
  SIGNAL parser_st17 : BIT;
  ...
  SIGNAL parser_st1 : BIT;

BEGIN
  PART : BLOCK (clock='1' AND NOT clock'STABLE)
  BEGIN
    p1 <= GUARDED Np1 WHEN reset='0' ELSE '1';
    p2 <= GUARDED Np2 WHEN reset='0' ELSE '0';
    ...
    p28 <= GUARDED Np28 WHEN reset='0' ELSE '0';
    p29 <= GUARDED Np29 WHEN reset='0' ELSE '1';
    serpar_p3 <= GUARDED Nserpar_p3 WHEN reset='0' ELSE '0';
    ...
    serpar_p10 <= GUARDED Nserpar_p10 WHEN reset='0' ELSE '0';
    parser_p19 <= GUARDED Nparser_p19 WHEN reset='0' ELSE '0';
    ...
    parser_p18 <= GUARDED Nparser_p18 WHEN reset='0' ELSE '0';
  END BLOCK;
-- Dataflow description for transitions
  t1 <= NOT p2 AND linkin AND p1;
  ...
  t8 <= NOT parser_p18 AND p17 AND p12 AND (ivalid AND NOT p14);
  ...
  serpar_st7 <= serpar_p9 AND NOT serpar_p10;
  ...
  serpar_st1 <= serpar_p3 AND NOT serpar_p4;
  parser_st17 <= parser_p26 AND i7 AND NOT parser_p27;
  ...
  parser_st1 <= parser_p18 AND NOT parser_p19;
-- Dataflow description for next place markings
  Np1 <= t10 OR t3 OR (p1 AND NOT t1);
  Np2 <= t1 OR (p2 AND NOT t10 AND NOT t2);
  ...
  Np29 <= t7 OR (p29 AND NOT t3);
  Nserpar_p3 <= t2 OR (serpar_p3 AND NOT serpar_st1);
  ...
  Nserpar_p10 <= serpar_st7 OR (serpar_p10 AND NOT t3);
  Nparser_p18 <= t8 OR (parser_p18 AND NOT parser_st1);
  ...
  Nparser_p27 <= parser_st16 OR parser_st17 OR (parser_p27 AND NOT t9);
-- Output Signals Equations
  linkout <= t8 OR t5 OR parser_st1 OR parser_st3 OR ... OR parser_st17;
  iack <= t10;
  qvalid <= t4;
  shiftenable <= t2 OR serpar_st1 OR ... OR serpar_st7;
---- ASSERT Commands
-- Transitions in conflict
  ASSERT NOT (t10 AND t3)
    REPORT "output place p1 overflows (transitions t10,t3)"
    SEVERITY ERROR;
  ASSERT NOT (t9 AND t6)
    REPORT "output place p17 overflows (transitions t9,t6)"
    SEVERITY ERROR;
  ASSERT NOT (t8 AND t5)
    REPORT "t8 and t5 are in conflict (input place p17)"
    SEVERITY ERROR;
-- No Enabled Transitions
  ASSERT NOT (t1='0' AND .. AND serpar_st7='0' AND ... )
    REPORT "Petri Net may be deadlocked"
    SEVERITY WARNING;
END dataflow;
```

Figure 13: Generated data flow VHDL code for the link adaptor controller, including a `BLOCK` statement.

ations. If transitions `t3` and `t10` were simultaneously enabled, the place `p7` would be not safe. If both transitions were fired, output place `p1` would be marked twice, violating the desired PN safeness. Similar situations would occur for transitions `t6`, `t9` and transitions `t5`, `t8` with respect to place `p17`.

For simulation and synthesis the ALLIANCE package [31] was used, since the subset accepted by its tools includes all the constructs used in the SIPN descriptions, including `BLOCK` and `PROCESS` statements.

To simulate the system controller of the link adaptor the generated VHDL file is used together with a test vectors file. The ALLIANCE simulator was fed with those files, and no errors were detected.

The synthesis process can be completed by using the ALLIANCE synthesizer. This tool generates a VHDL structural description from a RTL VHDL specification.

7 Conclusions

A complete ECAD framework which allows the specification, analysis, animation, simulation and synthesis of a hierarchical SIPN-based controller has been presented. This new set of design tools support a structured intermediate rule-based specification of a parallel controller from an SIPN, as an alternative to write unstructured VHDL specification and verify its correctness. Since there is a direct correspondence between the verified PN described in CONPAR and the generated VHDL code, there is no need to formally validate the VHDL programs.

The main part of the framework is the CONPAR to VHDL compiler. It produces VHDL code at the RT-level, that is later used for both simulation and synthesis purposes. Despite the current limitations, the proposed framework shows to be an useful tool to design parallel controllers, particularly by minimizing the number of errors at the specification level, prior to any implementation.

Further research is currently undergoing to implement an appropriate graphic editor, to improve the analysis procedures for hierarchical models, and to include the data path into the PN animator.

References

[1] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–80, April 1989.

[2] James Pardey, Alain Amroun, Martin Bolton, and Marian Adamski. Parallel Controller Synthesis for Programmable Logic Devices. *Microprocessors and Microsystems*, 18(8):451–8, October 1994.

[3] T. Kozlowski, E.L. Dagless, J.M. Saul, M. Adamski, and J.K. Szajna. Parallel controller synthesis using Petri nets. In *IEE Proceedings - Comp. Dig. Tech.*, volume 142, pages 263–71, July 1995.

[4] James Pardey and Martin Bolton. Logic Synthesis of Synchronous Parallel Controllers. *Proceedings of the IEEE International Conference on Computer Design*, pages 454–7, 1991.

[5] R. Valette. Analysis of Petri Nets by Stepwise Refinements. *Journal of Computer and System Science*, (18):35–56, 1979.

[6] João M. Fernandes. Petri Nets and VHDL on the Specification of Paralell Controllers. Master's thesis, Dep. Informática, Universidade do Minho, Braga, Portugal, July 1994. (in Portuguese).

[7] Joel M. Schoen (Ed.). *Performance and Fault Modeling with VHDL*, chapter 3.3.2: Petri Net Models. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[8] Jean Mermet (editor). *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*. Kluwer Academic Publishers, Dordrecht, 1992.

[9] Shishir Agarwal. Thinking Petri Nets Through VHDL. In *1990 VHDL Fall User's Group Meeting*, pages 51–9, Oakland, U.S.A., October 1990.

[10] J.-M. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL Designer's Reference*, chapter 5: System Modeling, pages 129–45. Kluwer Academic Publishers, Dordrecht, Boston, London, 1992. ISBN 0-7923-1756-4.

[11] Louis Baker. *VHDL Programming with Advanced Topics*. John Wiley & Sons, 1993.

[12] H. Belhadj, L. Gerbaux, M.-C. Bertrand, and G. Saucier. Specification and Synthesis of Communicating Finite State Machines. In G. Saucier and J. Trilhe, editors, *Synthesis for Control Dominated Circuits*, pages 91–102, 1993.

[13] Camila Runesson. Generation of VHDL Code from Coloured Petri Nets. Master's thesis, Linköping Institute of Technology, Sweden, February 1993.

[14] Zebo Peng. Digital System Simulation with VHDL in a High-level Synthesis System. *Microprocessing and Microprogramming*, 35:263–70, 1992.

[15] Ivy Team. System Specs. Leaflet.

[16] G. Swaminathan, R. Rao, J.H. Aylor, and B.W. Johnson. A VHDL based Environment for System Level Design and Analysis. In *VHDL International Users Forum (Spring Conference)*, pages 110–6, May 1994.

[17] L.P. Benders and M.P. Stevens. Petri Net Modelling in Embedded System Design. In P. Dewilde and J. Vandewalle, editors, *CompEuro 1992: Computer Systems and Software Engineering*, pages 612–7, May 1992.

[18] Petru Eles, Krzysztof Kuchcinski, Zebo Peng, and Marius Minea. Synthesis of VHDL Concurrent Processes. In *Euro-DAC '94, European Design Automation Conference with Euro-VHDL '94*, pages 540–5, Grenoble, France, September 1994.

[19] S. Olcoz and J. M. Colom. A Colored Petri Net Model of VHDL. *Formal Methods in System Design*, 7(1/2):101–23, August 1995.

[20] IEEE. *EURO-DAC'95: European Design Automation Conference with EURO-VHDL'95 (Proceedings).* IEEE Computer Society Press, Brighton, Great Britian, September 1995.

[21] Manuel Silva and Robert Valette. Petri Nets and Flexible Manufacturing. In G. Rozenberg, editor, *Advances in Petri Nets 89*, volume 424 of *Lecture Notes in Computer Science*, pages 376–417. Springer-Verlag, Berlin, Germany, 1990.

[22] Marian Adamski. Parallel Controller Implementation using Standard PLD Software. In W. R. Moore & W. Luk, editor, *FPGAs.* Abingdon EE&CS Books, 1991.

[23] Marian Adamski and João L. Monteiro. Rule-based formal specification and implementation of Logic Controllers programs. In *IEEE International Symposium on Industrial Electronics, ISIE'95*, pages 700–5, Athens, Greece, July 1995.

[24] K. Bilinski, M. Adamski, J. Saul, and E. L. Dagless. Petri net based algorithms for parallel-controller synthesis. In *IEE Proceedings - Comp. Dig. Tech.*, volume 141, pages 405–12, November 1994.

[25] M. R. K. Patel. Random Logic Implementation of Extended Timed Petri Nets. *Microprocessing and Microprogramming*, 30:313–320, 1990.

[26] J. M. Fernandes, A. M. Pina, and A. J. Proença. Simulation and Synthesis of Parallel Controllers based on Petri Nets. In *VII SBAC-PAD Simpósio Brasileiro de Arquitetura de Computadores — Processamento de Alto Desempenho*, pages 481–92, Canela, Brazil, July 1995. (in Portuguese).

[27] Ichiro Suzuki and Tadao Murata. A Method for Stepwise Refinement and Abstraction of Petri Nets. *Journal of Computer and Systems Sciences*, 27(1):51–76, August 1983.

[28] João M. Fernandes, António M. Pina, and Alberto J. Proença. Concurrent Execution of Petri Nets based on Agents. In *1st Workshop on Object-Oriented Programming and Models of Concurrency within the XVI International Conference on Applications and Theory of Petri Nets*, Torino, Italy, June 1995.

[29] Josef Grosch and Helmut Emmelmann. A Tool Box for Compiler Construction. Compiler Generation Project, Report no. 20, GMD — Universität Karlsruhe, January 1990.

[30] R. Taylor. Transputer Communication Link. *Microprocessors and Microsystems*, 10(4):211–5, 1986.

[31] Alain Greiner and François Pêcheux. Alliance: A complete set of CAD tools for teaching digital VLSI design. In *3rd Eurochip Workshop on VLSI Design Training*, pages 230–237, Toledo, Spain, September 1992.

A The CONPAR language

The complete CONPAR language grammar is presented: the constructs of the language are described with some examples. The Backus-Naur Form (BNF) notation is used to show the language sintax.

A.1 Generic Structure

The generic structure of a CONPAR source file follows:

<ParallelController> ::=
<Header>
(<MacroPlace> | <MacroTransition>)*
<Part>+
[**.PREDICATEDESCRIPTION** <PredDescr>+]
.E

Remarks can be added to the source file and they can be nested to any level. To open a remark the string "< *" is used and the remark ends with the string "* >".

The language is not case-sensitive. To write the keyword "**.marking**" any of the following strings is allowed: "`.marking`", "`.Marking`", "`.MARKING`", "`.MaRkInG`". The same applies for any other keyword or identifier in the language.

A.2 Global Signals

Global signals represent all the signals that are available in any part of the controller. Those signals include the clock, the primary inputs and outputs and the predicates.

Sintax

<Header> ::=
.CLOCK <id>
[**.INPUT** <id>+]
[**.OUTPUT** <id>+]
[**.PREDICATE** <id>+]

A.3 Macronodes

Macronodes (macroplaces and macrotransitions) can be seen as procedures of an HLL and they let the controller to be hierarchically specified. The interfaces are the ports of a macroplace or macrotransition.

Sintax

<MacroPlace> ::=
.MACROPLACE <id> (<id>* , <id>*)
.INTERFACE <id> , <id>
<MacroNodeHeader>
.NET <Transition>+
[**.MOOREOUTPUT** <Moore_Output>+]
[**.PREDICATEDESCRIPTION** <PredDescr>+]
[<Initial_Marking>]

<MacroTransition> ::=
.MACROTRANSITION <id> (<id>* , <id>*)
.INTERFACE <id> , <id>
<MacroNodeHeader>
.NET <Transition>+
[**.MOOREOUTPUT** <Moore_Output>+]
[**.PREDICATEDESCRIPTION** <PredDescr>+]

A.4 Parts

The parts represent the several PNs that form the specification of a parallel controller, i.e. the various subcontrollers that together constitute the global controller. The parts can be interconnected or not interconnected. The places and transitions defined in parts are global, which means that places and transitions in any part should have distinct names.

Sintax

<Part> ::=
.PART <id>
<PartHeader>
.NET <Transition>+
[**.MOOREOUTPUT** <Moore_Output>+]
[**.PREDICATEDESCRIPTION** <PredDescr>+]
<Initial_Marking>

A.5 Transitions

For each transition in a part or macronode, there is a specification, which describes the input and output places, its guard and the Mealy outputs activated at that transition.

Sintax

<Transition> ::=
<label> **:** <PreConditions> |**-** <PostConditions> ;

<label>: the transition being specified.
<PreConditions>: list of input places and an optional guard (separated by '*'). The guard should be specified as a single name. If the guard is just a single input signal (negated or not), that signal can be directly used. If the guard is written with more than one signal (inputs and places), the respective logical expression should be indicated by a predicate (see subsection A.7). Whenever there are inhibitor or

enabling arcs connecting a place to a given transition, its specification must be written with a predicate (to distinguish from “normal” input places). An inhibitor arc is specified by negating its input place, while an enabling arc is specified by simply referring the source place from where it comes.
<PostConditions>: list of output places and Mealy output signals (only activated when the transition is enabled).

A.6 Moore Outputs

For each place, a specification of the activated output signals must be considered. If no output signal is activated at a given place, no specification should be written.

Sintax

<Moore_Output> ::=
<id> |- <Active_Outputs> ;

<id>: the place where the outputs are activated.
<Active_Outputs>: list of the active output signals (separated by a ’*’) when the respective place is marked.

A.7 Predicates

When a transition’s guard contains an expression with enabling or inhibitor arcs or with more than one input signal, a predicate must be used in the transition specification.

Sintax

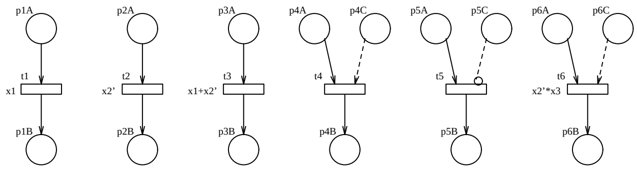
<PredDescr> ::=
<id> = <Logic_Function> ;

<Logic_Function>: a logical expression with input signals and places. The operators in table 1 can be used to built a logical expression:

Symbol	Meaning
*	logical AND
+	logical OR
!	logical NOT

Table 1: Symbols used to define logical expressions for predicates.

To group logical expressions, parenthesis, ’(’ and ’)’, can be used. Note the examples in Fig. 14, assuming that `xi` are input signals, `pi` are places, `ti` are transitions, and `predi` represent predicates.




```
t1: p1A *  x1   |- p1B;
t2: p2A * !x2   |- p2B;
t3: p3A * pred3 |- p3B;
t4: p4A * pred4 |- p4B;
t5: p5A * pred5 |- p5B;
t6: p6A * pred6 |- p6B;
...
pred3 = x1+!x2;
pred4 = p4C;
pred5 = !p5C;
pred6 = (!x2*x3)*p6C;
```

Figure 14: Transitions of an SIPN and their respective code in ConPar.

For transitions `t1` and `t2`, predicates are not necessary because their guards contains a single input signal (inverted for transition `t2`). The other transitions need predicates. The transition `t3` has a guard whose logical expression involves two input signals. For transitions `t4` and `t5`, a predicate must be used, because there are an enabling arc with origin in place `p4C` and an inhibitor arc with origin in place `p5C`, respectivelly. Transition `t6` represents the most generic situation, when the transition guard is a logical expression with input signals, enabling and inhibitor arcs.

A.8 Initial Marking

For any part or macroplace, the places initially marked should be specified.

Sintax

<Initial_Marking> ::=
.MARKING <id>+

<id>: specifies any of the places initially marked.

A.9 CONPAR *Language Grammar*

<ParallelController> ::=
<Header>
(<MacroPlace> | <MacroTransition>)*
<Part>+
[**.PREDICATEDESCRIPTION** <PredDescr>+]
.E

<Header> ::=
.CLOCK <id>
[**.INPUT** <id>+]
[**.OUTPUT** <id>+]
[**.PREDICATE** <id>+]

<MacroPlace> ::=
.MACROPLACE <id> (<id>* **,** <id>*)
.INTERFACE <id> **,** <id>
<MacroNodeHeader>
.NET <Transition>+
[**.MOOREOUTPUT** <Moore_Output>+]
[**.PREDICATEDESCRIPTION** <PredDescr>+]
[<Initial_Marking>]

<MacroTransition> ::=
.MACROTRANSITION <id> (<id>* **,** <id>*)
.INTERFACE <id> **,** <id>
<MacroNodeHeader>
.NET <Transition>+
[**.MOOREOUTPUT** <Moore_Output>+]
[**.PREDICATEDESCRIPTION** <PredDescr>+]

<MacroNodeHeader> ::=
.PLACE <Node>+
.TRANSITION <Node>+
[**.PREDICATE** <id>+]

<Part> ::=
.PART <id>
<PartHeader>
.NET <Transition>+
[**.MOOREOUTPUT** <Moore_Output>+]
[**.PREDICATEDESCRIPTION** <PredDescr>+]
<Initial_Marking>

<PartHeader> ::=
[**.INPUT** <id>+]
[**.OUTPUT** <id>+]
.PLACE <Node>+
.TRANSITION <Node>+
[**.PREDICATE** <id>+]

<Transition> ::=
<id> **:** <Cond> (***** <Cond>)* **|-** <id> (***** <id>)* **;**

<Cond> ::=
<id>
| **!** <id>

<Moore_Output> ::=
<id> **|-** <id> (***** <id>)* **;**

<PredDescr> ::=
<id> **=** <Logic_Function> **;**

<Logic_Function> ::=
<id> <AuxLF>
| **!** <Logic_Function> <AuxLF>
| **(** <Logic_Function> **)** <AuxLF>

<AuxLF> ::=
+ <Logic_Function>
| ***** <Logic_Function>
| ε

<Initial_Marking> ::=
.MARKING <id>+

<Node> ::=
<id> <NodeAux>

<NodeAux> ::=
= <id> **(** <id>* **,** <id>* **)**
| ε