VHDL generation from hierarchical Petri net specifications of parallel controllers

J.M.Fernandes
M.Adamski
A.J.Proença

Indexing terms: VHDL, generation, Parallel controllers, Petri nets

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 adapter, 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 (PNs) [1] allow easy specification of co-operating subsystems and the use of formal validation methods, which are based on mathematical theory. These methods should be applied prior to the implementation phase and in parallel with the simulation phase, to minimise 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 transition 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 co-operating activities. A new software framework has been 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.

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.

The PN has been 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 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 visualised 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
- transition firings are synchronised 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. [1] (from [4]), presents an SIPN example, where xl are input signals and yl are output signals.
Some quasi-independent sub-PNs may require priority and synchronisation 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, as shown in Fig. 2, represented by a dashed line with a circle, connects a place to a transition and enables 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.

A synchronisation between two processes can be modelled by an enabling arc shown in Fig. 3. 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.

Structured programming concepts are also useful for dealing with the specification and modelling of complex or large controller systems. Hierarchical specification mechanisms can be introduced in macronodes (macroplaces or macrottransitions) [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 SIFN, consider the PN and the macroplace represented in Fig. 4a [6]. The resulting PN, after expanding the macroplace, is presented in Fig. 4b.

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.

An undesirable occurrence on an SIFN 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 syntactic and semantic correctness of the model to be tested. Experimental results, developed at Innos in a practical design, achieved a 90% area reduction and a 40% 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].

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

The PN modeling 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 controller design, is closely related to 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 supports 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.

Fig. 4 Hierarchy in SIPS

- a SPN with a macroplace; b exploding the macroplace

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 the CAMAD system. However, the PN was transformed into the classical FSM model, and the advantages of concurrency were lost.

Fig. 5 Undesirable occurrences on an SIPN

- a conflict; b overflow

Some of the most recent developments related to modelling and analysis were reported, for example, in [18–20].

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 given explicitly 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' nonprocedural 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 the PARIS system [2, 3, 24], and a new description format called the PN specification format (PNSF) was introduced.

The CONPAR specification format (in the Appendix) 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, and 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 describe sequencing explicitly, but sequences of operations could be easily derived by ordering the transition rules.

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

\[ <\text{label}>::=<\text{PreConditions}> \text{-} <\text{PostConditions}>;\]

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 state 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:

\[ t1:: p1 * x1 \text{ - } p2 * p3 * y1; \]

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 an 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...5), which are VHDL signals that model inputs to flip-flops. For transition t1, the relevant parts of the expressions are as follows:

\[ \text{Np1} ::= \ldots \text{OR} (p1 \text{ AND NOT} t1); \]

\[ \text{Np2} ::= t1 \text{ OR} \ldots; \]

\[ \text{Np3} ::= \ldots \text{OR} t1 \text{ OR} \ldots; \]

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:

\[ y1 ::= \ldots \text{OR} t1; \]

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 debugging, 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.

```
ENTITY controller IS
PORT (reset : IN BIT;
      st : Signal);
END controller;
```

```
ARCHITECTURE dataflow OF controller IS
  -- Place Signals
  SIGNAL p1 : BIT;
  SIGNAL p2 : BIT;
  SIGNAL p3 : BIT;
  SIGNAL p4 : BIT;
  SIGNAL p5 : BIT;
  SIGNAL p6 : BIT;
  -- Transition Signals
  SIGNAL t1 : BIT;
  SIGNAL t2 : BIT;
  SIGNAL t3 : BIT;
  SIGNAL t4 : BIT;
  SIGNAL t5 : BIT;
  SIGNAL t6 : BIT;
BEGIN
  PROCESS BEGIN
    WAIT UNTIL clock(EVENT) AND clock='1';
    IF reset='0' THEN
      p1 <= '0';
      p2 <= '0';
      p3 <= '0';
      p4 <= '0';
      p5 <= '0';
      ELSE
        p1 <= '1';
        p2 <= '1';
        p3 <= '1';
        p4 <= '1';
        p5 <= '1';
      END IF;
  END PROCESS;
  -- Dataflow description for transitions
  t1 <= NOT p2 AND NOT p1 AND t1 AND p1;
  t2 <= NOT p4 AND NOT p3 AND t2 AND p2;
  t3 <= NOT p5 AND t3 AND p3;
  t4 <= NOT p5 AND t4 AND p4;
  t5 <= NOT p1 AND p5 AND p4 AND (NOT t3);
  -- Dataflow description for next place markings
  Np1 <= t5 OR p1 AND (NOT t1);
  Np2 <= t1 OR p2 AND (NOT t2);
  Np3 <= t2 OR p3 AND (NOT t3);
  Np4 <= t2 OR p4 AND (NOT t4);
  Np5 <= t3 OR p5 AND (NOT t5 AND NOT t4);
  -- Output Signals Equations
  y1 <= p4 OR 0;
  y2 <= 0 OR 0;
  y3 <= 0 OR 0;
  y4 <= 0 OR 0;
  y5 <= 0 OR 0;
  y6 <= 0 OR 0;
END ARCHITECTURE;
```

The CONPAR description and generated VHDL code for a linear SIPN are shown in Fig. 7a; a CONPAR description in Fig. 7b. The text editor uses the CONPAR language notation. As an illustration, the CONPAR description for the SIPN in Fig. 1 is listed in Fig. 7a. 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 specified hierarchically, using expanded macroplaces.

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 analyzer 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 analyzer to verify the liveliness builds the reachability graph for an SIPN (a restricted reachability graph), as defined, for example, in [3, 14], and uses the PN structure. The liveliness of the SIPN is partially analyzed through the following tests:

(i) to check if there are source or sink places

(ii) to ensure that each marking has, at least, one successor marking

(iii) 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 outermost abstract level: when the PN is specified in a hierarchical way (see Fig. 4, for example), only the main PN is tested for validation. To overcome this limitation, the methods introduced by Viulet [5] and Suzuki et al. [27] will be considered. These methods show that it is possible to analyze the main PN and the macronodes separately, 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 interact with a running PN directly. 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. 7b, which includes a
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 adapter shown in Fig. 8 provides a full duplex interface between a transputer link and two unidirectional 8-bit buses. Data on the input bus (I0-I7) are multiplexed onto LinkOut whilst data on LinkIn are latched, via an 8-bit register, onto the output bus (Q0-Q7). A complete handshake protocol (IVaid, IAck, QValid, QAck) controls data transfer to and from the buses.

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

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

The system controller of the link adapter can be specified by the SIPP in Fig. 10. The design, a revision of an earlier prototype [2], is a concurrent Mslay machine comprising 29 places and 35 transitions. Two macroplaces—Figs. 11a and b—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 acknowledges packet detection.
- Places p28, p11, p12 and macroplace ParSer perform parallel-to-serial conversion, and LinkOut acknowledges packet sending.
- Places p13-p16 and p29 represent the second part of serial-to-parallel conversion: generating QValid output status signal, and waiting for an 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. 12.

Fig. 10  Main SPN for link adapter controller

Fig. 11  SPN for link adapter controller

Fig. 12  CONPAR code for link adapter controller

Fig. 13  Inhibitor arc to resolve a conflict situation
Using this notation to feed the properties analyser module, a potential conflict situation, similar to the one presented in Fig. 5a, is detected: transitions t5 and t8 share the input place p17. Note that these transitions are guarded by predicates Qack and 13 and, 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. 13.

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:

\[
\begin{align*}
p29 & \rightarrow p17: t8 \ t1 \rightarrow \text{parer} p29 \ p2 \\
& \rightarrow t8 \rightarrow \text{parer} p29 \ p1 \\
& \rightarrow t1 \rightarrow p29 p17 p12 p2
\end{align*}
\]

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. 14). 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 situations. If transitions t3 and t10 were enabled simultaneously, the place p7 would be not safe. If both transitions were fired, output place p1 would be marked twice, violating the desired PN safety. 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 SISP descriptions, including BLOCK and PROCESS statements.

To simulate the system controller of the link adapter, 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 an RTI VHDL specification.

7 Conclusions

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

The main part of the framework is the COMPAR 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 is shown to be a useful tool for designing parallel controllers, particularly by minimizing the number of errors at the specification level, prior to any implementation.
9.4 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 have distinct names.

Syntax:
<Part> ::= .PART <id>::= .NET <Transition>+ .MOOREOUTPUT <Moore Output>+ .PREDICATEDescription <PredDescr>+ .Initial_Marking

9.5 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.

Syntax:
<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 used directly. 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 9.7). Whenever there are inhibitor or enabling arcs connecting a place to a NET, a 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).

9.6 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.

Syntax:
<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.

9.7 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.

Syntax:
<PredDescr> ::= .id::= <PredicateFunction>:

<Logic Function>: a logical expression with input signals and places. The operators in Table 1 can be used to build a logical expression.

Table 1: Symbols used to define logical expressions for predicates

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>logical AND</td>
</tr>
<tr>
<td>*</td>
<td>logical OR</td>
</tr>
<tr>
<td>!</td>
<td>logical NOT</td>
</tr>
</tbody>
</table>

To group logical expressions, parenthesis, (' ' and '), can be used. Note the examples in Fig. 15, assuming that xi are input signals, pi are places, ti are transitions and predi represent predicates.

Fig. 15 Transitions of an SIFN and their respective code in CONPAR

For transitions t1 and t2, predicates are not necessary because their guards contain 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 p5C and an inhibitor arc with origin in place p5C, respectively. Transition t6 represents the most generic situation, when the transition guard is a logical expression with input signals, enabling and inhibitor arcs.

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

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

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

9.9 CONPAR language grammar
<ParallelController>::= .Header(<Macroplace> | <MacroTransition>)* .Part>+
