SPECIFICATION AND VERIFICATION OF CONCURRENT SYSTEMS IN CESAR

J.P. Queille and J. Sifakis
Laboratoire IMAG, BP 53X
38041 Grenoble Cedex, France

Abstract :

The aim of this paper is to illustrate by an example, the alternating bit protocol, the use of CESAR, an interactive system for aiding the design of distributed applications.
CESAR allows the progressive validation of the algorithmic description of a system of communicating sequential processes with respect to a given set of specifications. The algorithmic description is done in a high level language inspired from CSP and specifications are a set of formulas of a branching time logic, the temporal operators of which can be computed iteratively as fixed points of monotonic predicate transformers. The verification of a system consists in obtaining by automatic translation of its description program an Interpreted Petri Net representing it and evaluating each formula of the specifications.

1. INTRODUCTION

The aim of this paper is to illustrate by an example the use of the system CESAR for the analysis of the properties of parallel systems.
CESAR is a system for aiding the design and integration of distributed applications. Its input language is a high level language, inspired from CSP [Hoare 78], for the algorithmic description of systems of communicating sequential processes. CESAR allows a progressive validation during the design process by considering two complementary aspects in a description :

- coherence in data manipulation (static characteristics of data and exchanged variables, visibility and access rights...)
- validation of the dynamic behaviour of a description with respect to its specifications.

Behavioural analysis of a system described by a program in the input language is based on the study of a representation of it in terms of Interpreted Petri Nets (IPN). Figure 1 illustrates the general principle of the system CESAR : given an algorithmic description of a system by a program in a high level language, a model representing some aspects of the described functioning is obtained by automatic

translation. This model (an IPN) is treated by an analyzer in order to verify the conformity of the described system to given specifications. Specifications are a set of formulas of a branching time logic and express correctness properties which must be satisfied by the system. Using branching time logic instead of linear time logic as it has often been done [Gabbay 80] [Lamport 80] [Manna 81], is one of the peculiarities of our approach. It is shown that in this logic it is possible to compute

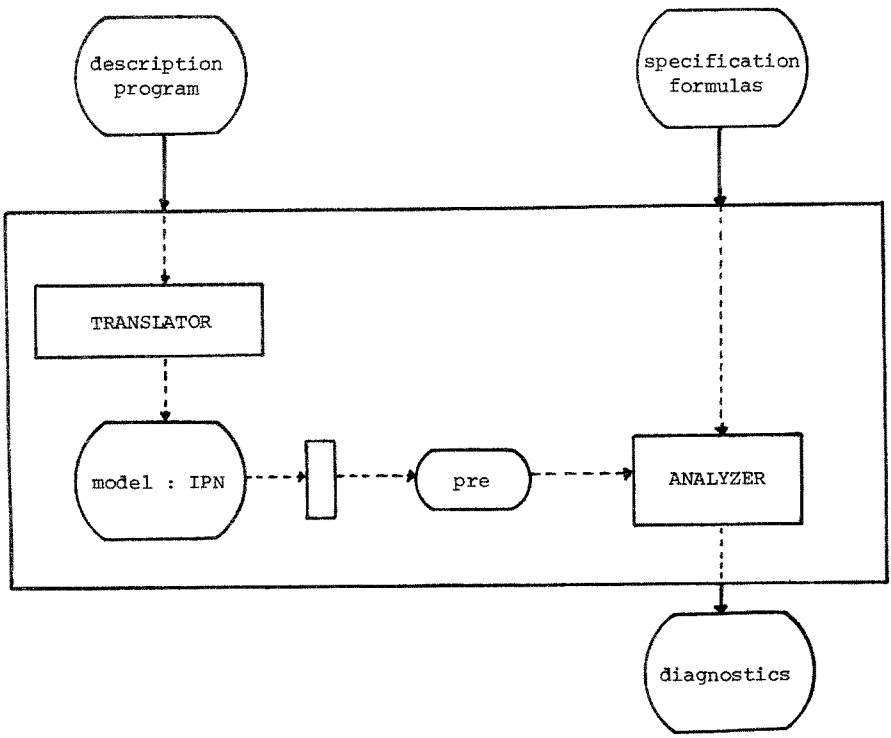

Figure 1

iteratively the interpretation of temporal operators as fixed points of monotonic predicate transformers.

Our approach presents some similarities to these followed in [Jensen 79] [Lauer 75] as far as the use of Petri nets as a model for the semantical analysis is concerned. The example considered throughout this paper is the alternating bit protocol. We have chosen this example because protocol modelling and verification is one of the principal application domains for CESAR. Furthemore, as protocols have been the object of many studies and especially the alternating bit protocol [Bartlett 69] [Bremer 79]

[Schwartz 81] [SIGPN 81], a rather precise comparison between the different approaches can be done.

This paper is organized in four parts. In part 2, the features of the description language are given and an illustration of its use for the description of the AB-protocol. After presenting the specification language, a part of the specifications of this protocol is given (part 3). In part 4 are exposed the analysis principle applied in CESAR and the theoretical results on which it is based.

2. DESCRIPTION IN CESAR

2.1. The description language

A system is described as a set of communicating sequential processes. Communications are declared as names of "exchanged variables". Exchange is done by rendez-vous between two processes, the one executing an output operation !V:=exp and the other an input operation ?V where V is the exchanged variable. The process executing the input operation has a local copy of the exchanged variable also denoted by V (not preceded by ?).

In addition to exchanged variables, processes have internal variables (which cannot be used for communication). Internal and exchanged variables are typed. Usual standard types and type constructors are available but the user can also introduce non-specified types for which it is not necessary to make manipulation rules explicit.

The basic statement of the language is the vectorial assignment. An input or output operation can be executed simultaneously with a vectorial assignment. We denote by nop the assignment whose right member is the identity function.

Besides the usual control structures, the CESAR description language provides the two following non-deterministic composed statements :

$$\underline{\text{if}}\ b_1 \rightarrow s_1\ //\ b_2 \rightarrow s_2\ //\ \ldots\ \underline{\text{fi}}$$

$$\underline{\text{do}}\ b_1 \rightarrow s_1\ //\ b_2 \rightarrow s_2\ //\ \ldots\ \underline{\text{od}}$$

where the b_i's are boolean conditions (guards) and the s_i's are sequences of statements. Their meaning is the following.

- IF : wait until one of the conditions is true and execute the corresponding sequence of statements.
- DO : repetition of an IF statement until a statement EXIT is encountered during the execution of some s_i.

For both of these constructs, if more than one conditions are true, the choice is non-deterministic. If a statement s_i begins with an input or output operation, the condition "the exchange can be executed" (i.e. the rendez-vous is possible) implicitely strengthens the guard b_i. The interpretation of the IF and DO constructs are

the same as the interpretation of the WHEN and CYCLE statements in [Brinch Hansen 78].

2.2. Translation of description programs into interpreted Petri nets

Given a program in the input language, the translator generates an Interpreted Petri Net (IPN) representing the main aspects of its behaviour. It performs also the type verification and deletes the internal variables of non-specified types.
The IPN corresponding to a program is obtained by composing the IPN's representing its sequential processes. The translation method of the process uses a graph grammar, every rule of which is associated with a rule of the grammar of the description language [Queille 81].
An IPN is a Petri net with :
- a vector of variables X,
- a mapping associating with each transition of the net a guarded command $c_i \rightarrow a_i$ where c_i is a condition on X and a_i is a vectorial assignment $a_i = (X:=\alpha_i(X))$.

Functioning rules of an IPN are those of standard Petri nets, with the addition of the following rules :
- a transition can fire only when its associated condition is true,
- when a transition fires, its associated action is executed.

IPN's are a useful tool for representing parallel programs in a non-deterministic way [Keller 76]. They can be graphically represented by the corresponding Petri net, the transitions of which are inscribed by the associated guarded commands. By convention, the always true condition and the identity assignment can be omitted. Thus, if a transition has no inscription its firing rule is the same as in a standard Petri net.
The translation method is such that each net representing a sequential process is a safe state graph. The composition rule expresses the rendez-vous by merging transitions and so, it preserves safety of each process. This property is used by the analyzer in order to simplify predicate manipulations.

2.3 Example : The Alternating Bit Protocol

2.3.1 Presentation of the protocol

The Alternating Bit Protocol (AB-Protocol) introduced in [Barlett 69] to provide a reliable full-duplex transmission over half-duplex links, is a protocol where the control information of each transmitted message or acknowledgement is a single control bit which can be used to detect loss of messages or acknowledgements and recover from them. In this paper, we are not interested in transmission errors which at the protocol level are not distinguished from losses. Since this protocol is completely symmetrical we suppose transmission of data in a single direction and describe it by considering a Sender and a Receiver as follows :

The Sender sends messages to the Receiver, which answers by sending acknowledgements. The Sender associates with each message a control bit which takes alternating values. After sending a message, the Sender does not change the control bit and does not send the next message before the reception of the corresponding acknowledgement (an acknowledgement with the same control bit). To recover from loss of messages or of acknowledgements, the Sender awaits the acknowledgement during a finite delay (measured by an arbitrary local clock) and then repeats the same message (without changing the control bit).

The Receiver behaves symmetrically. After receiving a message, it sends an acknowledgement with the same control bit and then awaits the next message (with a control bit of alternate value). If the next message does not arrive within an arbitrary local delay, the Receiver repeats the previous acknowledgement.

If we assume that the line cannot loose all the messages and acknowledgements (i.e. the line is not cut), this protocol ensures the correct transmission of each message after a sufficient number of repetitions. Message duplication does not cause any problem because the protocol guarantees that any sequence of received messages with the same control bit are duplications of the same message and the bit is changed by the Sender for all new messages. Thus, the Receiver has just to skip all the messages of such sequences except the first one. Symmetrically, the Sender has to skip duplications of acknowledgements in the same way.

2.3.2 Description programs and IPN's for the AB-protocol

We introduce two non-specified types :

- data to represent the data part of the messages
- pattern to represent the pattern of bits which is recognized as an acknowledgement.

Using the standard type boolean, we can define both the type msg for the messages and the type ack for the acknowledgements as two structures :

```
type msg = ( MESSAGE : data ;
             B : boolean ) ;
type ack = ( ACKNOWLEDGEMENT : pattern ;
             B : boolean ) ;
```

The program for the sender is given in the following page (T means true ; ^ is the complementation operator).

```
process SENDER
   ( output M : msg ;
     input A : ack ) ;

X : data ;
Y : boolean := 0 ;                                  -- initial value

begin

                    loop
send:                  !M := (X, Y) ;               -- send the message
                       do
receiveack:               T -> ?A ;                 -- receive acknowledgement
                             if
acceptack:                      A.B = Y -> Y := ^Y ;    -- expected acknowledgment
                                           exit //
skipack:                        A.B ≠ Y -> nop          -- else skip
                             fi //
repeat:                   T -> !M := (X, Y)         -- repeat the message
                       od
                    end loop

end SENDER ;
```

The program for the Receiver is the following :

```
process RECEIVER
   ( input MM : msg ;
     output AA : ack ) ;

Z : boolean := 0 ;                                  -- initial value

begin

                    loop
                       do
receive:                  T -> ?MM ;                -- receive message
                             if
accept:                         MM.B = Z -> exit //     -- expected message
skip:                           MM.B ≠ Z -> nop         -- else skip
                             fi //
repeatack:                T -> !AA := ("ack", ^Z) -- repeat previous acknowledgement
                       od;
sendack:               !AA := ("ack", Z), Z := ^Z -- send acknowledgement
                    end loop

end RECEIVER ;
```

The transmission line is described by the two following processes :

```
process SENDTORECEIVE
   ( input M : msg ;
     output MM : msg ) ;

begin

                    loop
get:                   ?M ;                         -- message is sent
                          if
transmit:                    T -> !MM := M //       -- message is transmitted
loose:                       T -> nop               -- message is lost
                          fi
                    end loop

end SENDTORECEIVE ;
```

```
process RECEIVETOSEND
   ( input AA : ack ;
     output A : ack ) ;

begin

                    loop
getack:                ?AA ;                        -- acknowledgement is sent
                          if
transmitack:                 T -> !A := AA //       -- acknowledgement is transmitted
looseack:                    T -> nop              -- acknowledgement is lost
                          fi
                    end loop

end RECEIVETOSEND ;
```

Figure 2 presents the IPN obtained by translation of the description program.

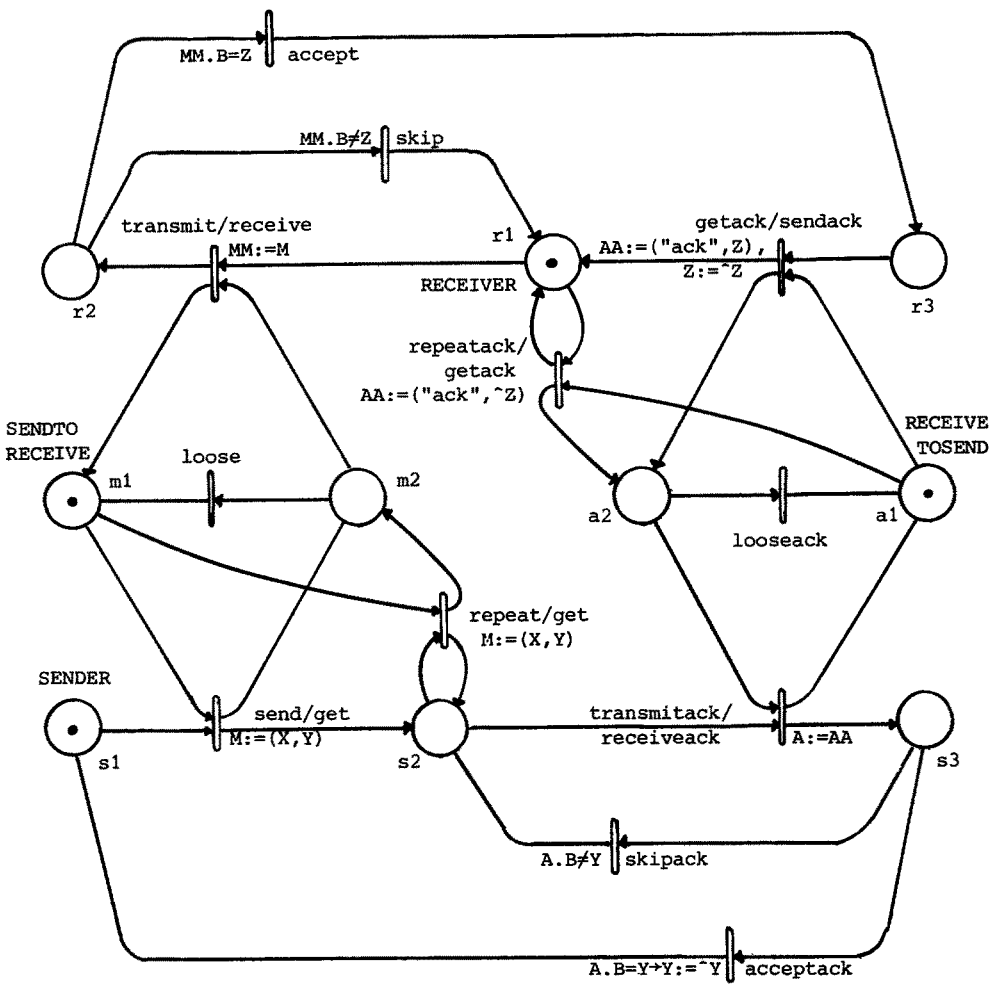

Figure 2

3. SPECIFICATION IN CESAR

3.1 The specification language

The specification language of CESAR is a branching time logic L [Lamport 80] [Rescher 71] constructed from a set of propositional variables F and the constants true, false, by using the logical connectives, $\neg$, $\wedge$, $\vee$, $\Longrightarrow$ and the unary temporal operators POT and INEV. The abbreviations ALL(f) and SOME(f) are used for respectively $\neg$POT($\neg$f) and $\neg$INEV($\neg$f).

The formulas of L represent assertions about the functioning of a given system if we consider that propositional variables represent predicates on its state and give a precise meaning to the operators POT and INEV. In order to do this, we consider transition systems as a model for L since IPN's can be given a semantics in terms of them [Keller 76].

A transition system is defined as a doublet S = (Q,$\rightarrow$) where Q is a set of states and $\rightarrow$ is a binary relation on Q ($\rightarrow \subset Q \times Q$). The relation $\rightarrow$ represents the actions or transitions of the system : $q \rightarrow q'$ means that there is an action executable from q which after its execution leads to a state q'. An execution sequence from a given state q_0 is a sequence s of states such that if s is finite then its last element q_t is a sink state (i.e. $\neg\exists q''(q_t \rightarrow q'')$). In order to simplify the notations we take s(k) to be equal to the k-th element of s if it is defined ; if not, we take $s(k)=\omega$ where ω represents some ficticious non accessible state adjoined to Q such that $\neg\exists q \in Q(q \rightarrow \omega)$. Thus, relation $s(0) \overset{k}{\rightarrow} s(k)$ is satisfied iff $s(k) \neq \omega$. The set of all the execution sequences from a state q will be denoted by EXq.

Given L and a transition system S=(Q,$\rightarrow$) we define an interpretation of L as a function $||$ associating to each formula of L a truth-valued function of the system state in the following manner :

- $\forall f \in F \quad |f| \in [Q \rightarrow \{tt,ff\}]$ where $[Q \rightarrow \{tt,ff\}]$ is the set of the unary predicates on Q
- $\forall q \in Q \quad |\underline{true}|\ (q) = tt$
- $\forall f \in L \quad |\neg f|(q) = tt$ iff $|f|\ (q) = ff$
- $\forall f_1, f_2 \in L \quad |f_1 \wedge f_2|(q) = tt$ iff $|f_1|\ (q) = tt$ and $|f_2|\ (q) = tt$
- $\forall f \in L \quad |POT(f)|(q) \equiv \neg\exists s \in EXq\ \neg\exists k \in \mathbb{N}\ [q \overset{k}{\rightarrow} s(k)$ and $|f|\ (s(k))]$
- $\forall f \in L \quad |INEV(f)|(q) \equiv \forall s \in EXq\ \neg\exists k \in \mathbb{N}\ [q \overset{k}{\rightarrow} s(k)$ and $|f|\ (s(k))]$

Obviously, |POT(f)| represents the set of the states q of S such that there exists an execution sequence starting from q containing a state satisfying |f|. We say that |POT(f)| is the set of the states from which some state of |f| is potentially reachable. In the same way, |INEV(f)| is the set of the states from which |f| is inevitably reachable in the sense that every execution sequence starting from a state of this set contains a state satisfying |f|.

The interpretation of the dual operators ALL and SOME is,

$$|ALL(f)|(q) \equiv \forall s \in EXq\ \forall k \in \mathbb{N}\ [q \xrightarrow{k} s(k) \text{ implies } |f|\ (s(k))]$$

$$|SOME(f)|(q) \equiv \exists s \in EXq\ \forall k \in \mathbb{N}[q \xrightarrow{k} s(k) \text{ implies } |f|\ (s(k))]$$

Remark that if the state q of a transition system satisfies |ALL(f)| then all the states of all the execution sequences from q verify |f|. Also, if a state q satisfies |SOME(f)| then there exists some execution sequence from q such that all its states verify |f|.

The properties of a branching time logic similar to L have been studied in [Ben-Ari 81] where a decision procedure and a complete deduction system are given.

3.2 Example

In this section we give examples illustrating the use of the specification language for expressing system properties. The formulas of this languages are constructed from the following set of propositional variables :

- propositional variables representing predicates on the variables of the description program (only program variables of specified types are considered),
- the propositional variable Init which characterizes the set of all the possible initial (control and data) states,
- propositional variables on the control of the system referring to names of actions (labels) defined in the description program ; for each labelled action a, the propositional variables enable a and after a are introduced such that |enable a| and |after a| characterize respectively the set of the states from which this action can be executed and the set of the states reached just after the termination of this action. The following abbreviations are used :
  - enable $(a_1,\ldots,a_k) = \bigvee_{i=1}^{k}$ enable a_i, where $\{a_1,\ldots,a_k\}$ is a set of actions,
  - enable P = enable A(P) where P is a process the set of the actions of which is A(P),
  - after $(a_1,\ldots,a_k) = \bigvee_{i=1}^{k}$ after a_i, where $\{a_1,\ldots,a_k\}$ is a set of actions.

Obviously, a large number of properties can be formulated concerning the behaviour of a system. For methodological reasons, it is interesting to classify the most important of them as this has already been done for linear time logic in [Gabbay 80] [Lamport 80] and [Manna 81]. Hereafter we introduce three families of properties and give specifications of the AB-protocol in terms of them.

Invariant properties

Invariant properties express the fact that a predicate P, constructed by using only logical operators, is always true.

They are formulas of the type : Init ==> ALL(P)

In the case of the AB-protocol such formulas can be used to express :

* Init ==> ALL(after(send,repeat) ==> (M.B=Y))
  i.e. after the emission of a message the value of the control bit emitted M.B is equal to the control bit Y of the SENDER.
* Init ==> ALL(after receive ==> (MM=M))
  i.e. after the reception of a message the value of the received message MM is equal to the value of the emitted message M, i.e. the line does not modify the transmitted information.
* Init ==> ALL(after receive ==> (MM.B=Y))
  i.e. after reception of a message the value of the received control bit MM.B is equal to the value of the control bit of the SENDER at the same time.

Liveness properties

These properties express the fact that an action can always be executed.

- Liveness of an action a : from every state q, successor of a state satisfying Init, there exists an execution sequence of EXq containing a state which enables a. This is expressed by the formula : Init ==> ALL POT enable a
- Liveness of a set of actions $\{a_1,\ldots,a_k\}$: each one of the actions a_i is live. This can be expressed by the formula : Init ==> $\bigwedge_{i=1}^{k}$(ALL POT enable a_i).

Or, by distributivity of ALL with respect to $\wedge$: Init ==> ALL($\bigwedge_{i=1}^{k}$ POT enable a_i).

- Absence of deadlock for a set of actions $\{a_1,\ldots,a_k\}$: from every state q, successor of a state satisfying Init, there exists an execution sequence of EXq which contains a state enabling at least one of the actions a_i. This is expressed by : Init ==> ALL POT enable $(a_1,\ldots,a_k)$.

Some interesting liveness properties of the given example are (starting from the weakest ones) :

* Init ==> ALL POT enable (SENDER), i.e. absence of deadlock for the SENDER
* Init ==> ALL POT enable (RECEIVER), i.e. absence of deadlock for the RECEIVER
* Init ==> ALL POT enable send, i.e. the action of emitting a new message is live
* Init ==> ALL POT enable accept, i.e. the action of receiving a new message is live.

Properties of response to an action

They are properties expressing the fact that an action b is a consequence of an action a.

- Possible response : if an action a is executed then it is possible that an action b becomes executable. This is expressed by the formula :
  Init ==> ALL(after a ==> POT enable b)
- Inevitably possible response : if an action a is executed then necessarily b becomes executable. This is expressed by the formula :
  Init ==> ALL(after a ==> INEV enable b)

Some interesting properties of this family for the given example are :

* Init ==> ALL (after (send,repeat) ==> POT enable receive)
  i.e. the line from the SENDER to the RECEIVER is able to transmit messages.
* Init ==> ALL (after(sendack,repeatack) ==> POT enable receivack)
  i.e. the line from the RECEIVER to the SENDER is able to transmit acknowledgements.
* Init ==> ALL(after(send,repeat) ==> INEV enable receivack)
  i.e. after sending a message the SENDER waits for an acknowledgement.
* Init ==> ALL[(after accept ==> INEV(enable sendack)) ∧
  (after sendack ==> AA.B=MM.B)]
  i.e. when the RECEIVER receives a new message, it will send the corresponding acknowledgement.

4. PROVING SPECIFICATIONS IN CESAR

4.1. The results used by the analyser

In this paragraph we present the basic theoretical results used by the analyser. The method consists in iteratively computing fixed points of predicate transformers obtained from the IPN under study. Fixed points are precisely the interpretations of temporal operators as it is shown by the following results proved in detail in [Sifakis 79].

Let S = (Q,→) be a transition system. It is convenient to identify any unary predicate on Q with its characteristic set. $(2^Q,\cup,\cap,\bar{\ })$ represents the lattice of predicates and $[2^Q \to 2^Q]$ the set of the internal mappings of 2^Q (predicate transformers). For $f,g\in[2^Q \to 2^Q]$, $f\cup g$, $f\cap g$, $\bar{f}$, $\tilde{f}$ and Id denote the functions $f\cup g = \lambda p.\ f(P)\cup g(P)$, $f\cap g = \lambda P.\ f(P)\cap g(P)$, $\bar{f} = \lambda P.\ \overline{f(P)}$, $\tilde{f} = \lambda P.\ \bar{f}(\bar{P})$, $Id = \lambda P.\ P$.

We also introduce the notations :

$$f^* = Id\cup f\cup f^2\cup\ldots\ f^k\cup\ldots = \bigcup_{i\in\mathbb{N}} f^i$$

$$f^x = Id\cap f\cap f^2\cap\ldots\ f^k\cap\ldots = \bigcap_{i\in\mathbb{N}} f^i$$

Definition 1 : Given S = (Q,→) a transition system, $P\in 2^Q$ and $q\in Q$, we define the predicate transformer pre : $pre\ P(q) \equiv \exists q'(q\to q'$ and $P(q'))$.

Proposition 1 :

Let f be a formula of L, S = (Q,→) a transition system such that → be image-finite and $||$ an interpretation of L in S.

a) $|ALL(f)| = \tilde{pre}^x|f|$

b) $|SOME(f)| = (Id\cap(pre\cup\tilde{pre}))^x|f|$

Proposition 2 :

Let f be a formula of L, S = (Q,→) a transition system such that → be image-finite and $||$ an interpretation of L in S.

a) $|POT(f)| = pre^*|f|$

b) $|INEV(f)| = (Id \cup pre \cap \tilde{pre})^* |f|$

4.2 The principle of the verification method

According to the results of the preceding section, it is possible to compute iteratively the interpretation of the temporal operators. We present hereafter the principle of the verification method applied by the analyzer :

Let f be a formula to be verified on a given program PROG and N the IPN obtained by translation from PROG. Denote by $F = \{f_1,\ldots,f_n\}$ the set of the propositional variables occuring in f.

- Associate a boolean variable with each place of N.
- For each *after* $a \in F$, express $|$*after* a$|$ as a predicate on these variables (if necessary, N is transformed by adding new places).
- For each *enable* $a \in F$, express $|$*enable* a$|$ as a predicate on the boolean control variables and program variables.
- Express *Init* as a predicate representing the set of all possible initial states (knowing the initial marking of the net and the initial values of program variables).
- Reduce N without transforming the places which are involved in the expression of the predicates of $|F| = \{|f_1|,\ldots,|f_n|\}$. Reducing N consists in applying transformation rules preserving the property expressed by f in order to obtain an IPN of less complexity.
- Compute the predicate transformer pre associated to the reduced IPN and then, the interpretation of temporal operators following the evaluation order imposed by the formula f. During these computations simplification rules are applied, taking advantage of the fact that sequential processes correspond to state graphs. Given that there is no criterion on the speed of the convergence of these iterations, the user can impose a maximum number of iterations.
- If some iterative computation yields no result within the acceptable number of iterations then the analyzer fails to give an answer. If not, it evaluates $|f|$: the property described by f is verified iff $|f|(q) = tt$ for every state q.

4.3 Example

For the AB-protocol, the liveness property *Init* ==> ALL POT *enable* (SENDER) is verified by computing successively :

1) *Init* $= s_1 m_1 a_1 r_1 \overline{yz}$. Intersection operators are omitted. The boolean variables s_i, m_i, a_i, r_i, represent the fact that the places with the same name have a token (see figure 2).
2) *enable*(SENDER) $= s_1 m_1 \cup s_2 m_1 \cup s_2 a_2 \cup s_3$
3) the interpretation of POT *enable* (SENDER) as the limit of : $P_{k+1} = P_k \cup pre(P_k)$ with

P_0 = enable (SENDER).

The following relations are invariants generated by the translator expressing the fact that each process is a safe state graph and they are used to simplify the boolean expressions computed by the analyzer :

$s_1\bar{s}_2\bar{s}_3 \cup \bar{s}_1 s_2 \bar{s}_3 \cup \bar{s}_1 \bar{s}_2 s_3 = T$ (T is the always true predicate)

$r_1\bar{r}_2\bar{r}_3 \cup \bar{r}_1 r_2 \bar{r}_3 \cup \bar{r}_1 \bar{r}_2 r_3 = T$

$m_1\bar{m}_2 \cup \bar{m}_1 m_2 = T$

$a_1\bar{a}_2 \cup \bar{a}_1 a_2 = T$

The first step of the computation gives, $pre(P_0) = s_3 \cup r_2 \cup r_3 \cup a_2 \cup m_2$. Thus,

$P_1 = P_0 \cup pre(P_0) = T$ and POT enable (SENDER) = T.

4) ALL POT enable (SENDER) = T

5) [Init ==> ALL POT enable (SENDER)] = T (the property is verified).

5. CONCLUSION

We have tried to illustrate with an example, the AB-protocol, the analysis method applied in CESAR.

This method is based on the idea of translating the description of a system, given in some high-level formalism, into a model for which there exists a verification theory. This approach presents the advantage, on the one hand of abstracting from all the details which are not relevant to the verification of the behaviour (for example, data represented by variables of non-specified types), on the other hand, of displaying the control structure (invariants, for instance). In particular, the translation into a Petri net gives the possibility of naming control points which makes the expression of the properties easier.

The language of the formulas allows the expression of a great number of foundamental properties (invariant properties, liveness properties, properties of response to an action). The use of such a language is interesting from a methodological point of view as it provides the possibility of classification and comparison of the properties according to various criteria. Also, the representation of properties by formulas using temporal operators leads to mechanizable proofs provided that a method for obtaining from a given description the associated predicate transformer pre be given. Computing fixed points of monotonic functions is, from a practical point of view, a central problem and it determines the limitations of our approach. Appart from the limitations of theoretical nature (non-decidability of the "interesting" system properties) serious problems appear when applying iterative methods which require the manipulation, simplification and comparison of predicates on many variables. For this reason, the current version of CESAR can verify formulas with variables of type boolean, enumerated and integer with known bounds, only.In order to simplify computations, the analyzer

encodes the enumerated and bounded integer variables so that it manipulates only boolean variables ; this coding is completely transparent to the user.

However, in spite of these simplifications the problems due to the complexity of the analyzed system remain crucial. We intend to increase the efficiency of the applied method by working in the following directions :

- Use of methods for approximating fixed points of monotonic operators in a lattice [Cousot 78] [Clarke 80],
- Reduction of the complexity of the iterative computations by decomposing global assertions into a set of local assertions,
- Study of a methodology of description since the possibility of proving a property greatly depends on the way the description is built.

REFERENCES

[Bartlett 69] K.A. BARTLETT, R.A. SCANTLEBURY and P.T. WILKINSON "A note on reliable full-duplex transmission over half-duplex links" CACM, Vol. 12, N°5, May 1969, pp. 260-261.

[Ben-Ari 81] M. BEN-ARI, Z. MANNA and A. PNUELI "The temporal logic of branching time" Proc. 8th Annual ACM Symp. on Principles of Programming Languages, Jan. 1981, pp. 164-176.

[Bremer 79] J. BREMER and O. DROBNIK "A new approach to protocol design and validation" IBM research report RC 8018, IBM Yorktown Heights, Dec. 1979

[Brinch Hansen 78] P. BRINCH HANSEN "Distributed Processes : A concurrent programming concept" CACM, Vol. 21, N°5, Nov. 1978, pp. 934-941.

[Clarke 80] E.M. CLARKE Jr. "Synthesis of resource invariants for concurrent programs" ACM Trans. on Progr. Languages and Systems, Vol. 2, N°3, July 1980, pp. 338-358.

[Cousot 78] P. COUSOT and N. HALBWACHS "Automatic discovery of linear restraints among variables of a program" Proc. 5th ACM. Symp. on Principles of Programming Languages, Tucson, Ariz., 1978, pp. 84-96.

[Gabbay 80] D. GABBAY, A. PNUELLI, S. SHELAH and J. STAVI "On the temporal analysis of fairness" Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages, Jan. 1980, pp. 163-173.

[Hoare 78] C.A.R. HOARE "Communicating Sequential Processes" Comm. ACM 21-8, August 1978, pp. 666-667.

[Jensen 79] K. JENSEN, M. KYNG and O.L. MADSEN "A Petri net definition of a system description language" Semantics of Concurrent Computation in LNCS, Springer Verlag, July 1979, pp. 348-368.

[Keller 76] R.M. KELLER "Formal verification of parallel programs" Comm. ACM 19, 7 (July 1976), pp. 371-384.

[Lamport 80] L. LAMPORT ""Sometime" is sometimes "not never" - On the temporal logic of programs" Proc. of the 7th Annual ACM Symp. on Principles of Programming Languages, Las Vegas, Janv. 1980, pp. 174-185.

[Lauer 75] P.E. LAUER and R.H. CAMPBELL "Formal semantics of a class of high level primitives for coordinating concurrent processes" Acta Informatica 5, pp. 297-332 (1975).

[Manna 81] Z. MANNA and A. PNUELI "Verification of concurrent programs : The temporal framework" Intern. Summer School, Theoretical Foundations of Programming Methodology, Munich, July 1981.

[Queille 81] J.P. QUEILLE "The CESAR system : An aided design and certification system for distributed applications" Proc. 2nd Int. Conf. on Distributed Computing Systems, April 1981, pp. 149-161.

[Rescher 71] N. RESCHER and A. URQUHART "Temporal Logic" Springer Verlag, Vienna, 1971.

[Schwartz 81] R.L. SCHWARTZ and P.M. MELLIAR-SMITH "Temporal logic specification of distributed systems" Proc. 2nd Int. Conf. on Distributed Computing Systems, April 1981, pp. 46-454.

[Sifakis 79] J. SIFAKIS "A unified approach for studying the properties of transition systems" Research Report RR N° 179, IMAG December 1979 (Revised December 1980), to appear in TCS January 1982.

[SIGPN 81] Special Interest Group : Petri nets and related system models. Newsletter N° 7, Feb. 1981, p-. 17-20.