ANALYSING SEMANTICS OF CONCURRENT HARDWARE SPECIFICATIONS

L.Ya. Rosenblum and A.V. Yakovlev
Computing Science Department
Leningrad Electrical Engineering Institute
Prof. Popov Street 5, Leningrad 197022 USSR

Abstract -- Of concern here a characteristic issue of concurrent hardware specifications: their behavioural correctness is manifested by the compliance between the global specification of the order of actions in a system and the local operational cliches of composite modules on which these actions are performed, with no particular regard for any timing constraints such as clock rates. Weak and strong forms of compliance are elicited with greater emphasis on a stronger form, called coherence. A relation-based framework for the semantic analysis of specifications defined in a high-level parallel program notation is outlined. It is shown by examples that the rapid prototyping of semantic analysis functions can be achieved through using a Prolog programming environment.

1. Introduction

Parallel programming techniques are becoming widely used in the design of the present-day VLSI hardware whose internal behaviour is increasingly concurrent - circuit subcomponents operate in a compliant and self-timed manner. Using such formal models as Petri nets [1], CSP [2], CCS [3], path expressions [4], trace structures and their program notation [5], temporal logic [6] etc. provides more or less effective media for proving various properties of concurrent behaviour, e.g., liveness, boundedness, delay-insensitivity, data-independence, composability with respect to a given class to name but a few.

Classifying the description languages and described processes, or circuits, according to some structural or behavioural attributes helps us to arrive at such classes of models that are powerful enough in their descriptive capacity and, at the same time, practically usable from the algorithmic viewpoint. It is also significant to create mechanical inference tools using such axiomatic rules that define the concurrent behaviour semantics. This is the subject of top interest elsewhere [7].

In this paper we use a rather conventional syntax for specifying concurrent processes in the self-timed logic. Processes are defined on a set of atomic actions related with the changes of discrete variable values and with checking of these values.

The high-level programming notation is very suitable for both realising the concept of silicon compilation and generally for developing an interactive high-level design environment.

The circuit compilation always incorporates lexical, syntactic and semantic analyses as well as the object code generation in the form of a sequence of procedures building the circuit layout which depends upon a particular CAD environment and given VLSI technology.

We do not refer here to the code generation aspects, but rather pay our major attention to the problem of analysing the semantic correctness of the self-timed logic behaviour specifications.

First, we present a notion of the general form of correctness as a compliance between the global behavioural semantics and local operational cliches of self-timed components (e.g., discrete control variables) that are subjected to some or other operations by corresponding atomic actions orderred in the global specification. We then illustrate this notion with an example of checking the general semantic correctness for a simple system in which an asynchronous register is exposed to the read and write operations by a concurrent schedule.

Second, we formulate a motivation for a relation-based view upon self-timed hardware behaviour as this is the most adequate form for defining the correctness of a system where events are ordered in time by the causal relationships rather than by some clock mechanism [8]. The notion of correctness is reduced to such more specific characteristics as coherence and sign-compliance.

Third, we show how the relation-based semantics can be extracted from a high-level programming notation. It uses such relational attributes as "concurrent" (con), "alternative" (alt) and "sequential" (seq), which are associated with major control flow constructs from a subset of the context-free grammar.

The important issue about our approach to the semantic analysis is that it is syntax-oriented because the relations between atomic actions are computed on a parsing tree, which is presumed to be after the syntax analysis step.

Fourth, we discuss some aspects of the relational semantics verification with respect to the coherence property. This property is related to the requirement about conflict-free actions on the discrete variables which may represent self-timed components of various types, from simple binary control variables to status registers and state machines.

We end the paper with illustrating the suitability of Prolog for the relational semantics extraction and checking. Prolog has been effectively used for the construction of a prototype software that realises lexical, syntax and semantic analyses for hardware specifications made in the Ossip language [9]. These steps of the silicon compilation process have been implemented in our Prolog programming environment for a fairly short time of a M.Sc. level project.

An example of practical application of the relation-based approach has been chosen in proving the partial correctness of the protocol specification of the IEEE-896.1 Futurebus standard [10].

In the bus acquisition logic specification, some flaws relating to the coherence violation have been found and later corrected in the final draft.

2. Compliance as a general form of behavioural correctness

We presume that a self-timed system is a composition of a set of self-timed objects. Each object is of a particular type, e.g., register, buffer, state-machine, variable, wire, void signal etc. We restrict ourselves with a finite number of object types (this restriction is however not critical) and thereby with a finite number of the object's intrinsic operational cliches. The operational cliche of every new type of object is defined at the time of incorporating a new component definition into the design environment.

The behaviour of the self-timed system can be defined in terms of a high-level programming notation whose lowest operational level may be the so-called "atomic actions" level. From both syntactic and semantic viewpoints atomic actions are assumed to be the elementary operations - such as assignments, events, checks - that are locally permissible and can be performed on the objects of the above mentioned types. For example, the binary control variable x , having the type "Bit", can be subjected to actions +x (transition from 0 to 1) and -x (transition from 1 to 0). Due to the self-timed nature of the behaviour the operational order between atomic actions must be defined by means of causal relationships without any notion of a common clock, which usually determines the starting point for every action in synchronous systems.

By the operational cliche of a particular type we mean the semantics of its local behaviour defined on its interface with the outside system with respect to those actions that can be performed on the given object. For example, for the binary control variable the local operational cliche is a totally sequential ordering of events +x -x +x -x ... , which can be floored to, say, a regular expression in the form:

$$(+x; -x)^*$$

where ";" and "*" denote the concatenation and iteration operators respectively.

We can formulate our notion of operational correctness of a global specification of the system's behaviour through introducing the concept of compliance.

By the compliance we mean that if we identify all actions of the global behavioural specification with corresponding actions of each local operational cliche of the system object (in other words, if we introduce each component object into the system structure), then we obtain the composition which should meet certain correctness criteria. For example, the weak form of compliance, similar to a weak form of liveness or termination, may require only the existence of at least one, either infinite, as in the case of liveness, or leading from a given initial state to some goal state, in the case of termination, sequence which satisfies the "accomplished"-like goal of the global behaviour.

On the other hand, the stronger form of compliance may demand that the global operational semantics extracted from the specification is totally compliant with the local component cliches such that the projection of the global behaviour on the set of actions performed on each (or some) component(s) is identical to the local cliche(s) of that (those) components.

The above notion of operational compliance between the global specification and each local component cliche can be illustrated by the following example inspired by [11].

Let a system consist of a register and the environment which may include a number of concurrent processes organised in a global schedule of actions concerned with reading and writing the values from and to the register.

The register can store any value from a finite set $X(Reg) = \{x_1, x_2, \ldots, x_n\}$. The allowed actions on such a type belong to the set $Act(Reg) = \bigcup_{1\le i\le n} \{w.x_i, r.x_i\}$ where $w.x_i$ is the "write value x_i into the register", and $r.x_i$ is the "read value x_i from the register" operations.

We assume the "register" type to have the following operational cliche, defined in terms of a regular expression

$$(w.x_1;(r.x_1)^* ! w.x_2;(r.x_2)^* ! \ldots ! w.x_n;(r.x_n)^*)^*$$

where ! stands for the semantics of alternative. This cliche implies that the register value may be read as many times as needed, zero times inclusive, only if this value has been previously written into the register, but not overwritten with some other value.

Let us choose two possible global schedules which are highly concurrent. These schedules can be defined by regular expressions extended with the "," operator, having the semantics of concurrent execution of its operands:

(Sch1) $w.x_1;(w.x_2,w.x_3,r.x_2,(r.x_3;w.x_4));r.x_3$

(Sch2) $w.x_1;(w.x_2,w.x_3,r.x_2,r.x);r.x_3$

In order to check whether these two schedules are compliant with the register cliche we have to analyse the conditions for the strong and weak compliances. It is easily noticed that the strong compliance is not held because the global schedule does not have the mutual exclusion mechanism which could guarantee that only one value is attempted to be read or written at a time. On the other hand, the register must be provided with such a mechanism as it does not allow any concurrent read and write operations - it is a totally sequential object.

The weak compliance can be checked using some of the existing techniques. For example, we may refer to the trace theory [4], in which an attractive notion of weaving between two trace structures may be effectively exploited for such a check.

Let S1, S2 and R denote processes (prefix-closed structures of traces) generated by Sch1 and Sch2 and the register's cliche, respectively. We build S1 $\underline{w}$ R and S2 $\underline{w}$ R where $\underline{w}$ stands for the weave operator which is defined as follows.

If $X = < aX, tX >$ is a process where aX is a finite alphabet of events and $_*tX$ is a prefix-closed set of traces, $tX \in (aX)^*$, then the weave of two processes X and Y is the process

defined by

$$X \underline{w} Y = \langle aX \cup aY, \{ t \mid t \in (aX \cup aY)^* : t \upharpoonright aX \in tX \wedge t \upharpoonright aY \in tY \} \rangle$$

where $t\ A$ stands for the projection of trace t on alphabet A.

In our example, we have

$S1 \underline{w} R = \langle aS1, \{\varepsilon\} \rangle$, and

$S2 \underline{w} R = \langle aS2, \{\varepsilon, w.x_1\ r.x_1\ w.x_2\ r.x_2\ w.x_3\ r.x_3\} \rangle$

where ε is the empty trace. From this we may deduce that the first schedule is non-compliant and the second is weakly compliant to the register specification since there exists at least one non-empty trace in the trace set $t(S2 \underline{w} R)$. This trace accomplishes the whole required schedule.

From the above example with the register we may notice that the weak compliance can be sufficient as a form or degree of correctness only in such cases when the local component behaviour is powerful enough. In this example, the register allows that concurrent reads and writes are submitted by the environment, and it is capable for performing the mutual exclusion between them. However, in a different situation, say, in the case of defining a global specification of a group of control variables, we should demand stronger forms of compliance, one of which is further examined using the relation-based approach.

3. Motivation of relational semantics

The main characteristic of our design objects, self-timed discrete structures, is that they are composed of finite state components, or variables, whose states can be changed by causal dependencies defined by the specification of their behaviour.

Let a self-timed control logic circuit be represented as an interconnection of binary variables of the set $Z = \{z_1, z_2, \ldots, z_n\}$ with a set of allowed variable state changes $DZ = \bigcup \{+z_i, -z_i\}$. The behaviour of the circuit can be specified by labelled Petri net shown in Fig.1, in which the transitions are associated with the changes in DZ by a corresponding partial function, called the labelling function. This Petri net generates the marking diagram shown in Fig.2, which can be used for studying some properties related to the order of transition firings.

The need for establishing the order relationship between transitions follows from their semantics. Such semantics, the variable state changes, requires, first, that for every variable all changes of its state must be properly ordered with respect to the initial marking of the Petri net. In other words, there must be no reachable marking in which any two or more transitions labelled with the same variable are enabled. Such a feature will be referred to as coherence. Coherence guarantees that the specification is safe with respect to its operational semantics. The conceptual motivation of this feature is such that none of the pairs of parallel process paths may change the value of the same variable. The absence of coherence may however be interpreted in a different way for a group of parallel processes in which a simultaneous change of some variable has the "rendez-vouz" semantics [2,4], but in this case these concurrent changes must be labelled identically. In some other cases the concurrency between different changes of the same variable may not be an anomaly when, for example, the component corresponding to such a variable has an internal, built-in, facility which provides the mutual exclusion. This is quite similar to the idea of implementing critical regions in hardware by using arbitration circuits. In this text, we accept the "if non-coherent then non-safe" semantic attitude, in much the same manner as from the semi-modularity of an asynchronous circuit we deduce that this circuit will not be susceptible to race conditions [12]. It is also assumed that for the representation of parallel activities seeking for an action on the common object we should have in explicit form the mechanism for mutual exclusion, for example, by introducing a special syntax construct "arbitrator".

Another important feature, necessary for the self-timed specification to be correct, is the sign-compliance. We say that the specification is sign-compliant with respect to z_i if it is coherent with respect to z_i and the changes $+z_i$ and $-z_i$ are ordered in such a way that between any two changes of the form $+z_i$ $(-z_i)$ there must be at least one change of the form $-z_i$ $(+z_i)$. If the specification is sign-compliant with respect to all variables, it is referred to as sign-compliant specification.

For the given example we can easily see that the Petri net specification is non-coherent with respect to z_3, and non-sign-compliant with respect to z_2, z_3 and z_4. This fact can be formally established if we introduce the notions of sequential (seq) and concurrent (con) relations on the set of Petri net transitions. These relations can be defined by means of the marking diagram, which contains all necessary information about possible transition firing sequences of the net.

Two transitions t_i and t_j are said to be in the seq-relation, denoted as t_i seq t_j, if in all allowed firing sequences from the initial marking t_i precedes t_j, or t_j precedes t_i.

Two transitions t_i and t_j are said to be in the con-relation, denoted as t_i con t_j, if $\neg(t_i \text{ seq } t_j)$.

For the given Petri net, the triangular table shown in Fig.3 defines the seq- and con-relations on the set $T = \{t_1, \ldots, t_7\}$. Using these entries one can easily check the above properties of the specification.

This example shows how important for the establishing of the introduced features is to be able to compute the seq- and con- relations in the most effective way. The efficiency of finding the solution depends on the class of described processes. In this example we have used a formal technique provided by labelled Petri nets and the semantics of the marking diagram and firing sequences. The latter is usually called the interleaving semantics [13] of concurrency. In such a semantic view, one sees two transitions t_i and t_j concurrent if in the set of firing sequences one can find both a sequence with t_i preceding t_j and a sequence with t_j preceding t_i. The interleaving view is proved to be equally powerful as the

partial order semantics only for the class of the so-called distributive processes [14]. It is also true that the interleaving semantics is not preserved under the splitting of atomic actions into subactions that may be a serious disadvantage in some hierarchical design disciplines [13].

In the following section we discuss the semantic analysis procedure which does not require constructing either reachability graph (marking diagram) or all possible atomic action sequences.

4. Syntax and semantics of a behaviour specification language

In this section we shal use only a small part of the behavioural constructs of the Ossip hardware specification notation [9], specially chosen for our major purposes of dealing with the relational semantics of behavioural specifications. This semantics can be extracted from the program code in terms of the seq, con and alt (alternative) relations defined on the set of atomic actions. These relations can be subsequently used in the verification process whose major goal is to check the properties associated with the correct representation of self-timed circuits.

A fragment of the high-level notation grammar can be given by the following set of rules:

1) <behaviour> ::= BEHAVIOUR <composite_operator> END.
2) <composite_operator> ::= BEGIN <sequence_of_operators> END
3) <sequence_of_operators> ::= <operator> [; <sequence_of_operators>]
4) <operator> ::= <simple_operator> | <composite_operator>
5) <simple_operator>::= <if_operator> | <par_operator> | <assignment>
6) <if_operator> ::= IF <predicate_on_Id> THEN <operator> ELSE <operator> FI
7) <par_operator> ::= PARBEGIN <sequence_of_composite_operators> PAREND
8) <sequence_of_composite_operators> ::= <composite_operator> [;<sequence_of_composite_operators>]
9) <assignment> ::= Id := <expression_on_Id>

where Id stands for a lexical unit denoting a variable identifier.

These rules may seem rather trivial because they lack some other important constructs such as, for example, loop or event handling ones. But for the sake of clarity we restrict ourselves to them since the inclusion of, say, a LOOP ... POOL operator would demend the modification of operational semantics which will be concerned with another, different from the accepted, interpretation of the sequence and alternative relations.

With the three types of constructs, those given by rules 2, 6 and 7, we associate the relational attributes defining the corresponding relations between the lower level operators as follows.

For the <composite_operator> all suboperators between BEGIN and END are in the sequence relation, i.e., if BEGIN OP1; OP2; ... ; OPn END, then OPi seq OPj where $i \neq j$, $1 \leq i,j \leq n$.

For the operator of the PARBEGIN OP1; OP2; ...; OPn PAREND type all suboperators, which are themselves composite operators, are pairwise in the concurrent relation, i.e., OPi con OPj where $i \neq j$, $1 \leq i,j \leq n$.

For the conditional operator of the IF C THEN OP1 ELSE OP2 FI type, where C is an atom corresponding to the computation of a predicate on one or several identifiers, we have C seq OP1, C seq OP2 and OP1 alt OP2 where alt denotes the alternative relation between operators OP1 and OP2.

Note that all three relations introduced are irreflexive, symmetric and non-transitive.

We should also agree that any assignment or predicate will be called an atomic action, or simply an atom.

5. Analysis of relational semantics

The major goal of the first part of semantic analysis process is to construct the seq, con and alt relations between all atomic actions with their subsequent use in proving the corr-ctness properties. These relations can be extracted from the parsing tree, the result of the syntax analysys, which is formally represented as a pair $< N, anc1 >$ where N is a set of nodes representing the language (grammatical) constracts and anc1 is a non-transitive, asymmetric and irreflexive relation called the "the immediate (one-step) ancestor-descendant" relation on the set N, i.e. $anc1 \subset N \times N$. It should be noted that the parsing tree is used in a reduced form - the N set contains only the nodes associated with operators defined by the rules 2, 6 and 7,and with atoms.

For example, the following operator

IF A1 THEN A2 ELSE BEGIN A3;A4 END FI

where Ai (i = 1..4) are atoms has the reduced parsing tree shown in Fig.4. Non-terminal nodes n1 and n2 have the following relational attributes:

n1: seq [(n2,n3), (n2,n4)], alt (n3,n4)
n2: seq (n5,n6)

Bearing in mind that the language constructs are well-structured we can easily deduce that the following "Inheritance Axiom" holds.

Inheritance Axiom.

Let two nodes n_i and n_j be given such that they have common immediate ancestor n^*, i.e., n^* anc1 n_i and n^* anc1 n_j.

If n_i rel n_j, rel $\in R^j = \{seq, con, alt\}$, then for all the descendants of n_j, i.e., for every $n_j^i \in Anc(n_j) = \{n_k : n_j \text{ anc } n_k\}$ where anc is the transitive and reflexive closure of anc1, and for all the descendants of n_i, i.e., for every $n_i^j \in Anc(n_i)$, the relation n_i^j rel n_j^i is true.

This axiom shows that the descendants inherit the relation between their respective ancestors. We consider here only such semantics which satisfy the Inheritance Axiom (appearing to be a peculiar interpretation of Shakespeare's "Montekki-Capulet-ti" relationship).

It can be easily proved that this axiom is true for structured programs since they preserve the closedness of the relational semantics with respect to the structural decomposition or refinement of operators [13]. Furthermore, since for every pair of immediate descendants n_i and n_j of

some node n^* the latter is attributed with one of the three possible types of $rel \in R$ (the local completeness of operator semantics), then it may be asserted that for any pair of nodes in the reduced parsing tree belongs to exactly one of the relations rel' such that $rel' \in R' = R \cup \{anc\}$ (the global completeness of semantics). The uniqueness of the relation between any two nodes follows from the Inheritance Axiom and from the fact that for every node n^* the elements in R, attributed to n^*, are pairwise not intersected.

Thus if we obtain as a result of the syntax analysis the set N with the $anc1$ relation on it where each non-terminal $n \in N$ is associated with an attribute (a list of relations $rel \in R$ between immediate descendants), then it is clear that the relation between any two nodes n_i and n_j can be found in the following way. First, we check if these nodes are in the anc relation. This is checked after we have found the nearest common ancestor of n_i and n_j, which is a quite well-known problem, having several possible algorithmic solutions in literature. Let n^* denote such an ancestor. Second, we check in which relation are the immediate descendants of n^*, which are at the same time the ancestors of n_i and n_j, respectively. Formally the second step can be expressed as

$$n_i \ rel \ n_j = \exists n^*, n_i', n_j' : ((n_i' \ rel \ n_j') \wedge (n^* \ anc1 \ n_j') \wedge (n^* \ anc1 \ n_j') \wedge (n_i' \ anc \ n_i) \wedge (n_j' \ anc \ n_j) \wedge rel \in R .$$

The second part of the semantic analysis, the verification process, involves the formulation of a set of axioms defining those properties that the extracted semantics of the specification should possess. The problem of such a formulation is most difficult because of possible lack of detail in the correctness requirements. For the case of relational semantics our main interest is concerned with the structure of relations between the atomic actions which are defined on self-timed objects (variables, registers, flags etc.) as well as the asynchronous character of ordering of these actions. Hence we must check whether our global relational semantics is compliant with the local semantics of these objects.

One of the simplest forms of coherence, within the framework of this paper, would be expressed by the axiom

$$\forall x_i \in X \wedge \forall a_i, a_j \in Atom\_on(x_i) : \neg(a_i \ con \ a_j)$$

where X is the set of modifiable variables, and $Atom\_on(x_i)$ is the set of atoms involving variable x_i. Some, more fine, forms of coherence may also define relational configurations between atoms on the same variables, depending on whether these atoms are assignments or predicates, or wherther they are guarded by mutual exclusion facilities that are implicit, or even wether the variable is involved in the right or the left hand side of the assignment, or that variable is a wired-OR signal.

6. Implementing semantic analysis within a Prolog programming environment

It is quite suitable to use a Prolog programming environment for constructing a prototype software of hardware desgn tools. Prolog is convenient for writing rules of lexical, syntax and semantic analyses in a compact and executable form. The corrrection of grammar rules as well as the modification of correctness criteria can be done by local rewriting of corresponding axioms of the Prolog code.

We demonstrate here an example of a simple technique how a system of Prolog axioms can be written for the relations introduced above. Assume that the specification contains the following fragment:

```
IF  A1 THEN A2  ELSE
    PARBEGIN
       BEGIN  A3; A4  END;
       BEGIN  A5; A6  END
    PAREND
FI
```

in which Ai (i = 1..6) are atoms. As a result of the syntax analysis step we obtain the following list of elements of the relation $anc1(n1,n2)$, defining the parsing tree in the reduced form:

```
anc1(1,2).
anc1(1,3).
anc1(1,4).
anc1(4,5).
anc1(4,6).
anc1(5,7).
anc1(5,8).
anc1(6,9).
anc1(6,10).
```

where the integers in parentheses are associated with the tree nodes as follows:
1 - IF ... FI, 2 - A1, 3 - A2, 4 - PARBEGIN ... PAREND, 5 - BEGIN ... END, 6 - A3, 7 - A4, 8 - BEGIN ... END, 9 - A5, 10 - A6.

The other result of the syntax analysis is the information about the local semantics of operators, i.e., the relational attributes of the tree nodes which is obtained in the form of the following Prolog facts:

```
rel1(2,3,seq).
rel1(2,4,seq).
rel1(7,8,seq).
rel1(9,10,seq).
rel1(5,6,con).
rel1(3,4,alt).
```

The first part of the semantic analysis is concerned with computation of relations between any two atomic actions, which can be expressed in the following simplified form:

```
anc(N,N).
anc(N1,N2):- anc1(N,N2), anc(N1,N).
rel(N1,N2,anc):- anc(N1,N2), N1\=N2,!.
rel(N1,N2,anc):- anc(N2,N1), N1\=N2,!.
rel(N1,N2,R):- anc(N11,N1),
               anc(N12,N2),
               rel1(N11,N12,R).
```

where variables N,N1, N2, N11, N12 stand for the nodes in the parsing tree, and variable R denotes the title of the relation. In fact, this fragmant is capable to establish the relation between any two nodes in the tree, not only between atom ones. Provided that this code is loaded and compiled we may state a goal for finding a relation, say, between node 2 and node 6 in the following form

```
?- rel(2,6,R), print(R).
```

The program will produce the result - seq.

Since the analysis of the relational semantics of a self-timed circuit specification

defined on a set of discrete value objects is substantially based on the computation of the seq and con relations between atoms we should take into account a special characteristic of nodes, the fact of their atomicity. For the sake of clarity, we do not distinguish here the types of atoms, according to what has been said in the final paragraph of Section 5, and only define the atomic(N,Id_list) fact where apart from the atom number N we also have a list of identifiers involved both in the left and in the right hand sides of the atom as denoted by Id_list. Assume,for example, that atom A1 (node2) is a predicate, say, X=Y+Z, and atom A2 (node3) is an assignment, say, X:=0. Then their definition in the program database can be presented as facts

```
atomic(2, [x,y,z]).
atomic(3, [x]).
```

Here is the fragment which can check the fact of non-coherence with respect to a certain variable identified by id:

```
noncoherence(Id):- concurrent(N1,N2,Id),
                   write(N1), tab(3),
                   write(N2), tab(3),
                   write(Id), nl, fail.
concurrent(N1,N2,Id):- atomary(N1,Id),
                       atomary(N2,Id),
                       N1\=N2,
                       parallel(N1,N2).
atomary(N,Id):- atomic(N,Id_list),
                member(Id,Id_list).
member(X, [X|_]).
member(X, [_|Y]):- memeber(X,Y).
parallel(N1,N2):- rel(N1,N2,R), !, R=con.
parallel(N1,N2):- rel(N2,N1,R), !, R=con.
```

The collection of simple fragments of Prolog code presented here is of course just a hint on how we can test in a most rapid way those ideas discussed in the previous section. During the construction of a more or less versatile Prolog prototype for the semantic analyser we have to choose an adequate structural organisation of program modules and data files. Fig.5 shows a variant of such an organisation in which the files F1 through F8 contain the following data:

- F1 - the source code in the behaviour specification language,
- F2 - the list of lexical unities (internal representation code),
- F3 - the list of operator nodes and the anc1 relation (the reduced parsing tree),
- F4 - the list of local operator relations rel1 (relational attributes of non-terminal operators),
- F5 - the list of atoms with identifier lists,
- F6 - the list of nodes for which a complete set of relations is built,
- F7 - the list of global relations between all operator nodes,
- F8 - the result of verification, for example, the list of atoms and corresponding identifiers which do not satisfy the coherence condition.

7. Practical application of the method

The above technique and the Prolog implementation of the semantic analyser for self-timed logic specifications has been used in the checking of the intermediate draft of the BUS_ACQUISITION_LOGIC specification for the multiprocessor backplane standard IEEE-896.1 (Futurebus) [10,15]. We checked the coherence condition on the set of discrete variables. An error had been discovered in the specification of the PREEMPTION_AND_ERROR_CHECK operation. The atom which was a predicate involving variable STATUS and the other atom assigning a new value to STATUS were in the con relation which might have been resulted in a hazardous behaviour if implemented in that way. The error had been cured by inserting an additional flag variable into the path where the assignment took place. This flag is set to the true value in the case of executing the assignment of a new value to STATUS. The flag is then tested after two parallel paths are joined, and if it has been set to the true, the STATUS variable is assigned with a new value as required by the specification.

8. Conclusion

A relation-based approach to the semantic analysis of concurrent logic specifications provides rather suitable technique for combining the fine nature of a self-timed ordering of events in a system with those formal tools of reasoning about the correctness issues which are supported by a Prolog programming environment. In this paper we have only outlined in rather sketchy terms the way of checking the stronger forms of compliance between the global behavioural descriptions of the system and the local operational cliches of individual components involved in a global behavior. The coherence property, for example, demands that the global behaviour should preserve the structure of relations given on the set of atomic actions performed on a particular object in accordance with the object's operational cliche. In our examples here we required that all changes of values of a self-timed variable must be totally ordered in a sequence, thus not allowing the use of the same variable in concurrent actions. Future research in this field is quited open-doored, particularly, in creating the structure of various kinds of relational representations of coherence and weaker compliance forms for various types of asynchronous logic.

It is very important to distinguish between appropriate requirements of complying with these individual operational cliches in terms of the relational semantics which may further result in a comprehensive cliche library.

Another important issue is related to creating the flexible environment where various forms of compliance can be easiliy generated at the user-demand style, depending on the level of user's knowledge of the specified behaviour.

It is also desirable that the approach, and the techniques it implies have been applied to and tested on some more examples of real hardware design practice.

References

[1] J.L. Peterson, "Petri Net Theory and the Modelling of Systems,"Prentice-Hall, New York, NY, (1981).

[2] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, New York, NY, (1985).

[3] R. Milner, A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, Berlin,(1980).

[4] J.L.A. van de Snepscheut, Trace Theory and VLSI Design, Lecture Notes in Computer Science, Vol. 200, Springer-Verlag, Berlin, (1985).

[5] T.S. Anantharaman, E.M. Clarke, M.J. Foster, and B. Mishra, "Compiling Path Expressions into VLSI Circuits," Distributed Computing (1986), Vol.1, pp. 150-166.

[6] B. Mishra, and E.M. Clarke, "Hierarchical Verification of Asynchronous Circuits Using Temporal Logic," Theoretical Computer Science (1985), Vol. 38, pp. 269-291.

[7] Proceedings of the Seminar on Concurrency, Carnegie-Mellon University, Pittsburgh, PA, July 1984, Lecture Notes in Computer Science, Vol. 197, Springer-Verlag, Berlin, (1985).

[8] A.V. Yakovlev, "Designing Self-Timed Systems," VLSI Systems Design (September, 1985), pp. 70-90.

[9] E. Tarasova, The Ossip - a Language for Specifying Self-Timed Digital Systems, M.Sc. Thesis, Computing Science Department, Leningrad Electrical Engineering Institute, Leningrad, (February, 1987), In Russian.

[10] IEEE Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus, ANSI/IEEE Std 896.1 - 1987.

[11] J. Misra, "Axioms for Memory Access in Asynchronous Hardware Systems," Lecture Notes in Computer Science , Vol. 197, Springer-Verlag, Berlin, (1985).

[12] V.I. Varshavsky et al., Self-Timed Control of Concurrent Processes, Kluwer Academic Publishers, Dordrecht, (1989).

[13] L. Castellano, G. de Michelis, and L.Pomello, "Concurrency Versus Interleaving: an Instructive Example," Bulletin of the EATCS, (1987), Vol. 31, pp. 12-15.

[14] L. Rosenblum, A. Yakovlev, and V.Yakovlev, "A Look at Concurrency Semantics through Lattice Glasses," to appear in Bulletin of the EATCS.

[15] IEEE 896.1 Futurebus Working Group Mailing, IEEE Computer Society, (May-June 1985).

Illustrations

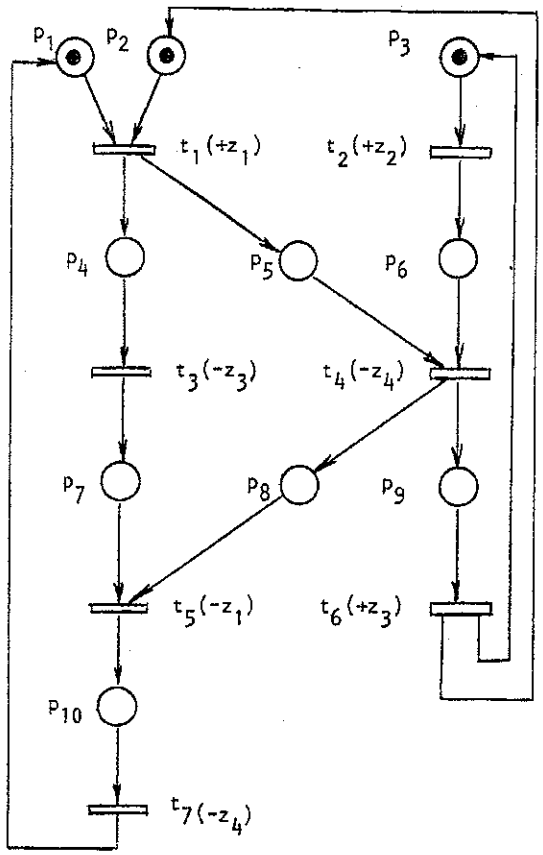

Figure 1.Specification of a self-timed circuit behaviour by labelled Petri net

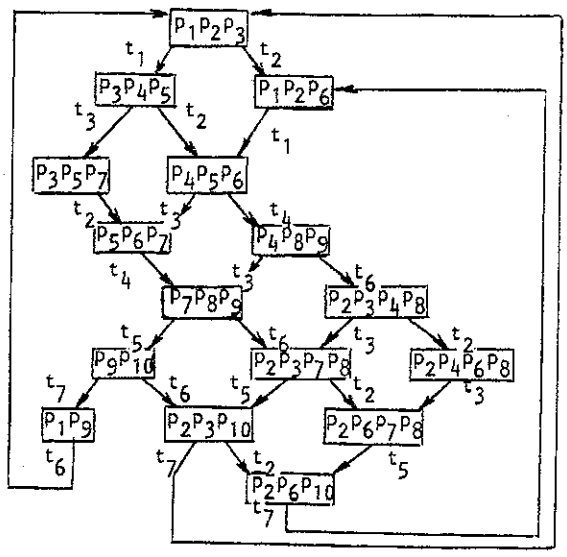

Figure 2. Marking diagram of Petri net shown in Fig.1

	t_7 $(-z_4)$	t_6 $(+z_3)$	t_5 $(-z_1)$	t_4 $(-z_4)$	t_3 $(-z_3)$	t_2 $(+z_2)$	t_1 $(+z_1)$
$t_1(+z_1)$	seq	seq	seq	seq	seq	con	-
$t_2(+z_2)$	con	seq	con	seq	con	-	
$t_3(-z_3)$	seq	con	seq	con	-		
$t_4(-z_4)$	seq	seq	seq	-			
$t_5(-z_1)$	seq	con	-				
$t_6(+z_3)$	con	-					
$t_7(-z_4)$	-						

Figure 3. Triangular table defining the seq and con relations

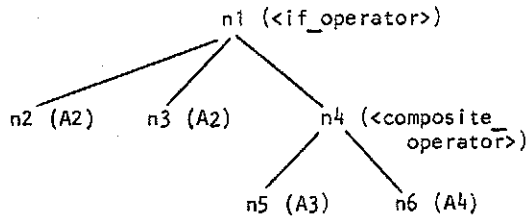

Figure 4. Example of specification parsing tree

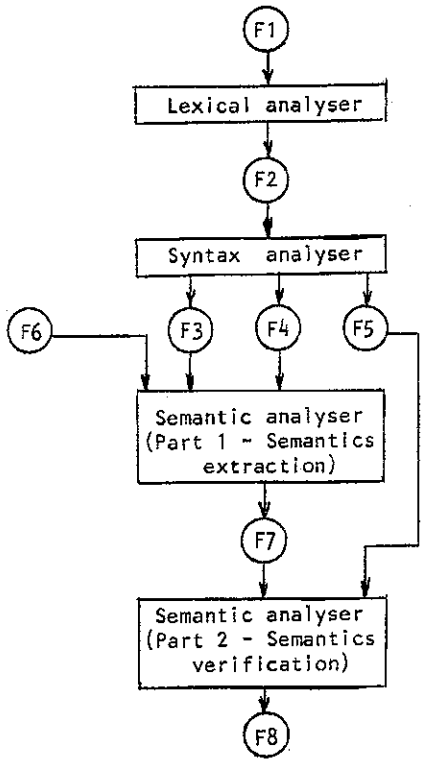

Figure 5. Example of organisation of Prolog prototype implementation