COMPUTING LABORATORY

A Relation-based Approach to Analysing Semantics of Asynchronous Hardware Specifications

A.V. Yakovlev
A Relation-based Approach to Analysing Semantics of
Asynchronous Hardware Specifications

A.V. Yakovlev

Abstract

Of concern here is a characteristic detail of asynchronous hardware
specifications: their behavioural correctness is determined by the
compliance between the global specification of the order between actions
in a system and all local operational cliches of composite modules on
which these actions are performed without any notion of timing
constraints. Weak and strong forms of compliance are elicited with
particular emphasis on a stronger form, coherence. A relation-based
framework for the semantic analysis of specifications given in a high-level
program notation are outlined. It is shown by examples that the rapid
prototyping of semantic analysis functions can be achieved through using
a Prolog programming environment.

Series Editor: M.J. Elphick

©1989 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

YAKOVLEV, Alexandre V.

A relation-based approach to analysing semantics of asynchronous hardware specifications. [By] A.V. Yakovlev

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1989.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 286)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 286

Abstract

Of concern here is a characteristic detail of asynchronous hardware specifications: their behavioural correctness is determined by the compliance between the global specification of the order between actions in a system and all local operational cliches of composite modules on which these actions are performed without any notion of timing constraints. Weak and strong forms of compliance are elicited with particular emphasis on a stronger form, coherence. A relation-based framework for the semantic analysis of specifications given in a high-level program notation are outlined. It is shown by examples that the rapid prototyping of semantic analysis functions can be achieved through using a Prolog programming environment.

About the author

Mr. Yakovlev was a visiting researcher in the Computing Laboratory from 1984 to 1985.

Suggested keywords

ASYNCHRONOUS AND SELF-TIMED HARDWARE SPECIFICATION
ATOMIC ACTIONS
COHERENCE
COMPLIANCE
HIGH-LEVEL PROGRAM NOTATION
OPERATIONAL CLICHE
PROLOG PROGRAMMING
RELATIONAL SEMANTICS
SEMANTIC ANALYSIS

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6424 621.38173
U.D.C. 519.682 621.3.049.771.14
A RELATION-BASED APPROACH TO ANALYSING SEMANTICS OF ASYNCHRONOUS HARDWARE SPECIFICATIONS

A.V. Yakovlev
Leningrad Electrical Engineering Institute
Computing Science Department
Leningrad 197022
U.S.S.R.

Abstract

Of concern here is a characteristic detail of asynchronous hardware specifications: their behavioural correctness is determined by the compliance between the global specification of the order between actions in a system and all local operational cliches of composite modules on which these actions are performed without any notion of timing constraints. Weak and strong forms of compliance are elicited with particular emphasis on a stronger form, coherence. A relation-based framework for the semantic analysis of specifications given in a high-level program notation are outlined. It is shown by examples that the rapid prototyping of semantic analysis functions can be achieved through using a Prolog programming environment.

keywords Asynchronous and self-timed hardware specification, high-level program notation, compliance, operational cliche, coherence, relational semantics, semantic analysis, Prolog programming, atomic actions.

1 Introduction

Parallel programming techniques are becoming widely exploited in the design of present-day VLSI hardware the internal behaviour of which is increasingly concurrent with respect to circuit sub-components operating in a compliant and asynchronous 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] and many others, provides more or less effective analysis media for establishing various properties of concurrent behaviour, e.g. liveness, safeness, boundedness, delay-insensitivity, data-independence, composability with respect to a given class etc.

The classification of description tools and described processes (or circuits) according to some structural or semantic attributes helps to arrive at such classes of models which are powerful enough in their descriptive capacity and at the same time acceptable from the viewpoint of algorithmic analysis. It is also significant to create mechanical inference tools using such axiomatic rules that define the concurrent behaviour semantics, something which is the subject of growing interest elsewhere [7].

*In 1984/85 on leave to Computing Laboratory, University of Newcastle upon Tyne, as a visiting researcher
In this paper we use a rather conventional syntax notation for specifying concurrent processes in self-timed logic. Processes are defined on a set of atomic actions related with changes of discrete variable values and with checking of these values.

The high level programming notation is extremely suitable for both realizing the concept of silicon compilation and generally for developing the interactive high level design environment.

The circuit compilation does necessarily incorporate lexical, syntactic and semantic analysis 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 to the code generation aspects here but rather pay major attention to the problem of analysing the semantic correctness of the self-timed logic behavioural specifications.

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

Second, we formulate a motivation for the relational semantics for a self-timed hardware behaviour which is the most adequate form for defining the correctness in a system where events are ordered in time by causal relationships rather than by some clock mechanism [8]. The notion of correctness is reduced to such particular properties as coherence and transition compliance.

Third, we show how the relation-based semantics can be extracted from a high level programming description. It is concerned with using relational attributes of 'concurrent', 'alternative' and 'sequential' types associated with main behavioural control flow constructs from a subset of the context-free grammar.

The important issue of such a semantic analysis is that it is syntax-oriented because the relations between atomic actions are computed on the parse tree which is presumed to be built after the syntactic analysis step.

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

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 prototype software that realizes lexical, syntactic and semantic analyses of descriptions made in a subset of STRICT-2 [9]. These steps of the silicon compilation process have been implemented in the Prolog programming environment in a fairly short period of time.

An example of a practical application of the relation-based approach has been chosen: the partial correctness checking of the protocol specifications of the IEEE-896.1 (Futurebus) standard, particularly the bus acquisition logic specification, where some flaws relating to the coherence violation have been found and corrected [10].
2 Compliance as a general notion of behavioural correctness of asynchronous hardware specifications

We presume that a self-timed system is a composition of a set of self-timed objects. Each object is supposed to be of a particular type, e.g. register, buffer, variable, wire, void signal etc., i.e. we restrict ourselves to a finite number of types (however, this restriction is not critical) and thereby to a number of objects intrinsic operational cliches. These cliches are defined at the time of incorporating a new module type definition into the design environment.

The behaviour of a self-timed system can be defined in terms of a high-level programming notation whose lowest operational level may be a so-called 'atomic actions' level. From both the syntactic and semantic viewpoints atomic actions are assumed to be elementary operators (assignments, events, conditional predicates) that are performed on the above objects. These actions must be locally permissible or appropriate with respect to the corresponding object types. For example, the binary control variable \( x \) having the type 'BIT' can be subjected to actions \( +x \) (transition from 0 to 1) or \( -x \) (transition from 1 to 0). The self-timed nature of the behaviour demands that the operational order between atomic actions is to be defined by means of causal relationships without any notion of a common clock mechanism determining the starting instance for every action.

By operational cliché of a particular object 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 cliché is a totally sequential order of events \(+x - x + x - x \ldots\) which can be floored to, say, a regular expression of the form:

\[(+x; -x)^\ast\]

where \( ; \) and \( \ast \) denote the concatenation and iteration operators, respectively.

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

By compliance we mean that if we identify all actions of the global behavioural specification with corresponding actions of every local operational cliché of the system structure object (in other words, if we introduce every modular object into the system structure) then we obtain the composition which will satisfy some given correctness criteria (or statement).

For example, the weak form of compliance, analogous to a weak form a liveness or termination, may require only the existence of at least one sequence, either infinite, as in case of liveness, or leading from a given initial state to some final goal state in case of termination, which satisfy 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 global specification is totally compliant with local module cliches such that the projection of the global specification on the set of actions performed on every (or some) module(s) is identical to the local cliché of that (those) module(s).

The above notion of operational compliance between the global specification and every local module cliché can be illustrated by following example inspired by [11].

Let a system consist of a register and the environment which may include a number of concurrent processes organized in a global schedule of actions concerned with reading and writing values in a 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 \( \text{Act}(\text{Reg}) = \bigcup_{1 \leq i \leq n} w.x_i, r.x_i \) where \( w.x_i \) and \( r.x_i \) are 'write value \( x_i \) into the register' and 'read value \( x_i \) from the register' operations.

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

\[
(\text{w}x_1; \ \text{r}x_1)^* | (\text{w}x_2; \ \text{r}x_2)^* | \cdots | (\text{w}x_n; \ \text{r}x_n)^* \]

where \(|\) stands for alternative semantics. 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 by some other value).

Let us choose two potential global schedules which are highly concurrent. These schedules can be defined by regular expressions extended with the comma (,) operator having the concurrency semantics:

\begin{align*}
\text{(SCH1)} & \quad w.x_1; (w.x_2, w.x_3, r.x_2, (r.x_3; w.x_4)); r.x_3 \\
\text{(SCH2)} & \quad w.x_1; (w.x_2, w.x_3, r.x_2, r.x_1); r.x_3
\end{align*}

In order to check whether these two schedules are compliant with the register cliche we have to analyse the conditions for strong and weak compliance. 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 written and read at a time, and it is highly concurrent whereas the register must have such a mechanism and does not allow any concurrency on its write and read ports (it is totally sequential).

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

Let \( S1, S2 \) and \( R \) denote trace structures generated by schedules \( \text{SCH1} \) and \( \text{SCH2} \) and the register cliche, respectively.

Let us build \( S1 \ w \ R \) and \( S2 \ w \ R \) where \( w \) stands for weave operator which is defined as follows.

If \( X = <aX, tX>\) is a trace structure where \( aX \) is a finite alphabet of events and \( tX \) is a set of traces, strings of symbols in \( aX \), i.e. \( tX \in (aX)^* \), then the weave of two trace structures \( X \) and \( Y \) is the process defined by

\[
X \ w \ Y = \langle aX \cup aY, \{t | t \in (aX \cup aY)^*: t[aX \in tX \land t[aY \in tY]\} >
\]

where \( t[A] \) stands for the projection of trace \( t \) on alphabet \( A \).

It is obvious that in our example

\[
S1 \ w \ R = \langle aS1, \emptyset \rangle \quad \text{and} \quad S2 \ w \ R = \langle aS2, \{w.x_1 \ r.x_1 \ w.x_2 \ r.x_2 \ w.x_3 \ r.x_3\} >
\]

from which we deduce that the first schedule is non-compliant at all, and the second is weakly compliant since there exists at least one non-empty trace in the trace set \( t(S2 \ w \ R) \). This trace accomplishes the whole required schedule.

From the above example with register we may notice that the weak compliance can be sufficient as a form or level of correctness only in such cases when the local module behaviour is flexible enough. In this example the register allows that concurrent reads and writes are submitted by the environment, and it is able to perform the mutual exclusion between them. But in another example, say, in case of defining the global specification of
a group of control variables we must often 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 modules, or, speaking from the algebraic point of view, variables, whose states can be changed by causal dependencies defined by the specification of their behaviour.

Let a self-timed control logic circuit is represented as a collection of binary variables from the set \( Z = \{z_1, z_2, \ldots, z_n\} \) with a set of allowed variable state changes \( DZ = \bigcup_i\{+z_i, -z_i\} \).

The behaviour of the circuit can be specified by the labelled Petri net shown in Fig. 1 in which transitions are associated with changes from \( DZ \) by corresponding partial 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 necessity of 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 of the initial marking of the Petri net. In other words, there must be no marking in which any two or more transitions labelled with the same variable are enabled. Such a property will referred to as coherence. The coherence feature guarantees that the specification is safe with respect of its operation semantics. The conceptual motivation of this feature is such that none of pairs of parallel process paths may change the value of the same variable. The coherence absence fact can, however, be interpreted differently for a group of parallel processes in which a simultaneous change of some variable value has the 'rendez-vous' semantics [2,4], but in this case these concurrent changes of the variable value must be identical. In some other cases the concurrency between different changes of the same variable value may not be an anomaly if, for example, the operational unit corresponding to such a variable has an implicit facility built-in which provides the mutual exclusion (an analogue of a critical region with arbitrator). In this text we accept the semantics 'if non-coherent then non-safe' in much the same manner as from the non-semimodularity of the asynchronous circuit it follows that this circuit may be susceptible to critical race conditions [12]. It is also assumed that for the representation of parallel activities seeking for an action on the common resource (variable) we should contain in explicit form the mechanism of mutual exclusion by using a special syntax construct, say, 'arbitrator'.

Another important property necessary for the self-timed structure specification to be correct is the sign-compliance. We say that the specification is sign-compliant with respect to variable \( z_i \) 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 Petri net specification of a control logic behaviour we can deduce that it is non-coherent (with respect to \( z_2 \)) and non-sign-compliant (with respect to \( z_2 \) and \( z_4 \), as well as to \( z_3 \)). This can be established if we introduce notions of sequential (seq) and concurrent or parallel (par) relations on the set of Petri net transitions. These relations can be defined by means of a graph of reachable markings constructed with respect to a given initial marking.

Two transitions \( t_i \) and \( t_j \) are in seq-relation \((t_i \textbf{ seq } t_j)\) if in all allowed firing sequences from the initial marking \( t_i \) precedes \( t_j \).
Two transitions $t_i$ and $t_j$ are in par-relation ($t_i \parallel t_j$) if $\neg (t_i \text{ seq } t_j \lor t_j \text{ seq } t_i)$ is true.

For the Petri net given in Fig. 1 the triangular table shown in Fig. 3 expresses relations seq and par on the set $T = \{t_1, t_2, \ldots, t_7\}$. From this table we can find that with respect to $z_1$ the specification is sign-compliant, with respect to $z_2$ it is incomplete (the reverse change $-z_2$ is missing) and hence non-sign-compliant, for $z_3$ it is non-coherent because $-z_3$ par $+z_3$, and for $z_4$ it is coherent, but not sign-compliant because a pair of transitions is labelled with $-z_4$ but not with $+z_4$.

This example shows the importance of establishing the introduced properties in order to be able to compute the relations seq and par in a most effective way. The effectiveness of obtaining the solution depends on the class of described processes. In this example we have used a formal technique provided by the labelled Petri nets and semantics of the graph of reachable markings and firing sequences, i.e. so-called interleaving semantics [13] of concurrent actions. In such a semantics a pair of transitions $t_i$ and $t_j$ are regarded as parallel if in the set of firing sequences we can find both a sequence in which $t_i$ precedes $t_j$ and a sequence where $t_j$ is before $t_i$. The interleaving semantics is proved to be rather limited, for example in [13] authors show that this semantics is not kept under the splitting of atomic actions into subactions which may be a serious disadvantage in some hierarchical design disciplines.

The following section will discuss a concept of semantic analysis which does not require to construct neither a reachability graph nor all allowed atomic action sequences.

This concept allows, from the one hand, to study the results of the semantics extraction process independently of the checking of the syntactic correctness and, at the same time, to exploit the result of the syntax analysis process - the parse tree - for computing the semantic relations on it.

4 Syntax and semantics of behaviour specification language

In this section we shall use only the small part of the STRICT-like behavioural constructs which is relevant to our purpose of dealing with relational semantics of behaviour specification. This semantics can be extracted from the program text in terms of sequential, parallel and alternative relations defined on the set of atomic actions. These relations can be subsequently used in the verification process whose main objective is to establish properties associated with the correct representation of self-timed circuits.

A fragment of the high-level programming notation grammar is given by the following group of rules:

1. $<\text{Behaviour}> ::= \text{BEHAVIOUR} <\text{composite}\_\text{operator}> \text{ END}.$
2. $<\text{composite}\_\text{operator}> ::= \text{BEGIN} <\text{sequence}\_\text{of}\_\text{operators}> \text{ END}$
3. $<\text{sequence}\_\text{of}\_\text{operators}> ::= <\text{operator}>|;<\text{sequence}\_\text{of}\_\text{operators}>$
4. $<\text{operator}> ::= <\text{simple}\_\text{operator}>|<\text{composite}\_\text{operator}>$
5. $<\text{simple}\_\text{operator}> ::= <\text{if}\_\text{operator}>|<\text{par}\_\text{operator}>|<\text{assignment}>$
6. $<\text{if}\_\text{operator}> ::= \text{IF} <\text{predicate}\_\text{on}\_\text{Id}> \text{ THEN } <\text{operator}> \text{ ELSE } <\text{operator}> \text{ FI}$
7. $<\text{par}\_\text{operator}> ::= \text{PARBEGIN} <\text{composite}\_\text{operators}> \text{ PAREND}$
8. $<\text{composite}\_\text{operators}> ::= <\text{composite}\_\text{operator}> |;<\text{composite}\_\text{operators}>$
9. \[\text{<assignment>} ::= \text{Id} ::= \text{<expression_on_Id>}\] where \text{Id} is a lexical unit denoting a variable identifier.

These rules may seem rather trivial because they lack some other important constructs, say, loop or event handling constructs. But for the sake of formal clarity we restrict ourselves to them, since the inclusion, for instance, of a loop operator will demand the modification of operational semantics which will be concerned with another interpretation of sequence and alternative relations, different from the 'acyclic' case.

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

For the composite operator all suboperators between the brackets BEGIN and END are in the sequence relation, i.e. if BEGIN OP1; OP2; ...; OPn END then OPi \text{seq} OPj if \(1 \leq i < j \leq n\).

For the operator of the type PARBEGIN OP1; OP2; ...; OPn PAREND all suboperators, which are composite operators themselves, are pairwise in the parallel relation, i.e. OPi \text{par} OPj if \(i \neq j, 1 \leq i, j \leq n\).

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

Due to the 'acyclic' nature of the chosen processes the \text{seq} relation (in contrast to \text{par} and \text{alt}) is asymmetric and, hence, directed, but it is transitive, whereas both \text{par} and \text{alt} are non-transitive and symmetric.

Any assignment or predicate in the if-operator will be referred as atomic action (or simply, atom).

5 Analysis of relational semantics

The main purpose of the first part of the semantic analysis process is to construct the relations \text{seq}, \text{par} and \text{alt} between all atomic actions with their subsequent use in discovering the correctness properties. These relations can be drawn from the parse tree which is formally represented as pair \(<N, \text{anc1}>\) where \(N\) is a set of nodes representing the language constructs and \text{anc1} is non-transitive, asymmetric and irreflexive relation 'Immediate (one-step) Ancestor-Descendant' (i.e. 'Father-Son') on set \(N\), \text{anc1} \subseteq N \times N.\) It should be noted that the parse tree is reduced in such a way that the set \(N\) contains only nodes associated with operators defined by rules 2, 6, 7 and with atoms. To put it differently, the complete parse tree containing all non-terminals is transformed to its reduced form by transitive elimination of all non-relevant nodes. Fig. 4 gives an example of the exclusion of non-terminal \(<\text{sequence_of_operators}>\) out of the tree. Moreover, keywords and delimiters which are the tree leaves are also excluded as such. All necessary information about relations between any two atoms can be elicited from the relation \text{anc1} and relational attributes associated with non-terminal nodes. The latter are constructed on the set of all immediate descendants of all non-terminal nodes, according to the accepted operator semantics.

For example, the following operator

\[
\text{IF } A_1 \text{ THEN } A_2 \text{ ELSE BEGIN } A_3; A_4 \text{ END FI}
\]
where $A_i$ ($i = 1...4$) are atoms, has the reduced parse tree shown in Fig. 5. Non-terminal nodes $n1$ and $n2$ have following relational attributes:

$$
n1 : \text{seq} \ (n2,n3),(n2,n4) \ , \ \text{alt} \ (n3,n4) ;
$$

$$
n2 : \text{seq} \ (n5,n6)
$$

Taking into account that the language constructs are well structured we can easily deduce that the following 'inheritance rule' holds.

**Inheritance Rule** Let two nodes $ni$ and $nj$ be given such that they have common immediate ancestor (father) $n*$, i.e. $n*$ $\textbf{anc1} \ ni$, $n*$ $\textbf{anc1} \ nj$. If $ni$ $\textbf{rel} \ nj$, $\textbf{rel} \in R = \{\textbf{seq,par,alt}\}$ then for ancestors of $ni$, i.e. for every $ni' \in \textbf{Anc}(ni) = \{nk : ni \ \textbf{anc} \ nk\}$ where $\textbf{anc}$ is a transitive and reflexive closure of $\textbf{anc1}$, and for all ancestors of $nj$, i.e. for any $nj' \in \textbf{Anc}(nj)$, the relation $ni' \\text{rel} \ nj'$ is true.

This rule shows that the descendants inherit the relation between their respective ancestors. We consider here only such semantics which satisfies the inheritance rule, appearing to be a peculiar interpretation of the relationship axiom of the 'Montekki-Capulettil' type.

It is easy to prove that this rule is true for structured programs because they preserve the closedness of the relational semantics with respect to the structural decomposition or refinement of operators [13]. Also, since for every pair of immediate descendants $ni$ and $nj$ of some node $N*$ the latter defines one of three possible types of $\textbf{rel} \in R$ (the local completeness of operator semantics) then it may be asserted that any pair of nodes in parse tree belongs to one and only one of relations $\textbf{rel}$ where $\textbf{rel} \in R' = R \cup \{\textbf{anc}\}$ (the global completeness of semantics). The uniqueness of the relation between any pair of nodes follows from the inheritance rule and from the fact that for every node $n*$ the elements in $R$ are pairwise not intersected.

Therefore, if we obtain as a result of the syntactic analysis the set $N$ with the relation $\textbf{anc1}$ where each non-terminal element $n \in N$ is associated with an attribute (a list of relations $\textbf{rel} \in R$ between immediate descendants), then it is easily seen that to compute the relation between any two nodes $ni$, $nj$ we need to do the following. First, we check whether these nodes are in the $\textbf{anc}$ relation. This is checked after we have found the nearest common ancestor of $ni$ and $nj$ (the least upper bound of the subset $\{ni,nj\} \subset N$). Let $n*$ denote such an ancestor. And, second, we check in what relation are the immediate descendants of $n*$ which are at the same time the ancestors of $ni$ and $nj$, respectively. Formally, we can express this step as follows:

$$
ni \ \textbf{rel} \ nj \equiv \exists n*, ni', nj' :$$

$$
((ni' \ \textbf{rel} \ nj') \land (n* \ \textbf{anc1} \ ni') \land (n* \ \textbf{anc1} \ nj')) \land$$

$$
(ni' \ \textbf{anc} \ ni) \land (nj' \ \textbf{anc} \ nj) \land \text{rel} \in R = \{\text{seq,par,alt}\})
$$

The second part of the semantic analysis, the verification process, involves the formulation of a set of axioms defining those properties that the specification should possess, or sometimes should be free of. The problem of such a formulation is most difficult because of possible lack of precision in the correctness requirements. For the case of relational semantics we are necessarily concerned with the structure of relations between atomic actions which are defined on self-timed objects (variables, registers, flags etc.) as well as the asynchronous character of matching between these actions.

It is therefore sensible to examine at a greater detail some conceptual aspects related to the formulation of the coherence property with respect to atomic actions and objects
whose type is somewhat wider than that of mentioned in Section 3.

In establishing the potential condition of a conflict between any two atomic actions involving a common variable (common resource) we may need to differentiate between possible types of these actions with respect to variables involved. Let us consider some cases of pairs of 'conflicting' atoms, i.e. atoms A1 and A2 for which A1 \textit{par} A2, and estimate the 'inadmissibility degree' for the asynchronous hardware implementation.

Let A1 and A2 are predicates involving at least one common variable. This situation is obviously not incorrect because the state of common variable does change neither in A1, nor in A2 but is used only for computation of the logic branching condition. The using of the same variable is equivalent to applying the value from a line (or a bus) to the input of a decision element.

Let A1 and A2 be atoms in which a variable X is assigned with a new value (X is in the left side of the assignment operator). This situation is generally not quite correct but the circuit designer may have intentionally admitted it. If the structural object corresponding to X has arbitrary switching delays for various switching occurrences then after executing these two atoms we obtain that X will acquire such a value which has been assigned by the latest of these atoms. Due to the asynchrony this value is indeterminate and therefore this situation is likely the result of incorrect specification. If both A1 and A2 assign the same value to X then, after they are completed in either order, X acquires one possible value. In some cases this situation may have sensible semantics, for example, when X models either wired-OR or gated-OR signal (merge-unit). The following example of a program specification fragment:

\begin{verbatim}
PARBEGIN
BEGIN

  IF Parity_bit = 1 THEN
    Indicate_error := 1 ELSE . . . FI

END;
BEGIN

  IF Error_message = 1 THEN
    Indicate_error := 1 ELSE . . . FI

END

PAREND
\end{verbatim}

depicts how two parallel paths incorporating the diagnostic checks eventually lead to setting the common error indicator to 1.

Assume that A1 and A2 are again assignments including variable X but in contrast to the above case one of the atoms has X in its right side whereas the other assigns to X some new value. Let the first atom have in its left side variable Y. This situation is by no means correct (non-safety) because variable Y may suffer from race condition concerned with application of the old value of X to the assignment of Y and then with spontaneous change of Y according to some new value of X.

The above differentiation of cases involving atomic actions sharing the same variable(s) shows how important to take into account the dependence of the verification criteria for the relational semantics of the specification upon various types of actions themselves as well as on types of objects (control variable, register, buffer etc.) subjected to these actions.
6 Implementation aspects of semantic analysis

The Prolog programming environment is quite suitable for constructing prototype software for a silicon compiler. Prolog is a convenient tool for writing rules for lexical, syntactic and semantic analyses in a compact and executable form. The correction of grammar rules as well as the modification of correctness criteria can be done by local rewriting of corresponding axioms of Prolog code.

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

\[
\text{IF A1 THEN A2 ELSE}
\begin{align*}
&\text{PARBEGIN} \\
&\text{BEGIN A3; A4 END;} \\
&\text{BEGIN A5; A6 END} \\
&\text{PAREND}
\end{align*}
\]

FI

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

\[
\text{anc1}(1,2), \\
\text{anc1}(1,3), \\
\text{anc1}(1,4), \\
\text{anc1}(4,5), \\
\text{anc1}(4,6), \\
\text{anc1}(5,7), \\
\text{anc1}(5,8), \\
\text{anc1}(6,9), \\
\text{anc1}(6,10).
\]

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

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

\[
\text{rel1}(2,3,\text{seq}). \\
\text{rel1}(2,4,\text{seq}). \\
\text{rel1}(7,8,\text{seq}). \\
\text{rel1}(9,10,\text{seq}). \\
\text{rel1}(5,6,\text{par}). \\
\text{rel1}(3,4,\text{alt}).
\]

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

\[
\text{anc}(N,N). \\
\text{anc}(N1,N2) :- \text{anc1}(N,N2), \text{anc}(N1,N).
\]
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 nodes in the parse tree variable and R denotes the title of the relation. In fact, this fragment is capable to establish the relation between any two nodes in the tree, not only atom nodes. Provided that this program is loaded we may state a goal for finding a relation between node 2 and node 6 in following form

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

The program will deduce the correct result (seq).

Since the analysis of the relational semantics of self-timed circuit specification which is defined on the set of discrete value objects is substantially based on the computation of relations seq and par between atoms we have to take into account a special characteristic of nodes, the fact about their atomicity. For the sake of simplicity we do not differentiate here types of atoms according to what has been mentioned in Section 5 and only define the fact atomic(N,Id..list) where apart from variable N a list of identifiers involved both in the left and in the right sides of the atom is given by Id..list. Assume for example that the atom A1 (node 2) is a predicate, say, X=Y+Z, and the atom A2 (node 3) 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]).

Shown below is the fragment which can check the fact of non-coherence with respect to some variable identified by Id:

noncoherence (Id):- concurrent(N1,N2,Id), write(N1), write(N2), 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\..]):- member(X,Y).
parallel(N1,N2):- rel(N1,N2,R), !, R=par.
parallel(N1,N2):- rel(N2,N1,R), !, R=par.

The collection of simple fragments of Prolog code presented above is of course only a hint how the ideas described in the previous section can be tested in a most rapid way. During the construction of a more or less versatile Prolog prototype of a semantic analyser we have to choose an adequate structural organization of program modules and files. Fig. 6 shows a variant of such an organization in which files F1 to F8 contain the following data:

F1 the source code in the behavioural specification language,
F2 the list of lexical units (internal representation code),
F3 the list of operator nodes and the relation anc1 (the parse tree reduced according to the compression operation),
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 method

The above technique and corresponding Prolog-implementation of the semantic analysis of self-timed circuit specifications has been used to check the intermediate draft of BUS_ACQUISITION_LOGIC specification for the multiprocessor backplane standard IEEE-896 (Futurebus) [10, 14]. We checked the coherence condition on the set of discrete variables. An error has been discovered in the specification of the PREEMPTION_AND_ERROR_CHECK_OPERATION. The atom which is a predicate involving the variable STATUS and the atom assigning a new value to STATUS were in the par-relation which might have resulted in a hazardous behaviour if implemented in such a form. The error has been cured by inserting an additional flag variable into the path where the assignment takes place. This flag is set to the true value in case of the necessity 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 in the true state the variable STATUS is assigned with a new value as required by the specification.

8 Conclusion

A relation-based approach to the semantic analysis of asynchronous logic behaviour specifications provides a 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 logic programming environment. In this paper we have only outlined in a rather sketchy terms the way of checking the stronger forms of compliance between the global behavioural descriptions at the entire circuit or system level and the local operational cliches of individual modules involved in a global behaviour. 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 according to the object cliche. In our examples we required that all changes of values of a self-timed variable must be totally ordered in a sequence not allowing the use of the same variable in concurrent actions. Future research in this field is quite open, particularly with respect to creating the structure of various kinds of relational representations of coherence and weaker compliance forms for typical sets of asynchronous objects. It is very important to differentiate the appropriate requirements to comply with these individual operational cliches in terms of relation-based semantics which may further result in the cliche library. Another important issue is to create the flexible environment where various forms of compliance can be easily generated at the user interface depending on the level of his or her knowledge of the specified behaviour. Prolog is very convenient for the rapid prototyping of such a framework.
Acknowledgements

The author would like to thank Mr. Albert Koelmans and his colleagues at the Computing Laboratory for their attention and support.

References


Fig. 1. An example of self-timed discrete system behaviour using labelled Petri nets.
Fig. 2. Marking diagram corresponding to the Petri net of fig. 1.
\[
\begin{array}{cccccccc}
  t7 & t6 & t5 & t4 & t3 & t2 & t1 \\
(-z4) & (+z3) & (-z1) & (-z4) & (-z3) & (+z2) & (+z1) \\
\end{array}
\]

<table>
<thead>
<tr>
<th>t1(+z1)</th>
<th>seq</th>
<th>seq</th>
<th>seq</th>
<th>seq</th>
<th>seq</th>
<th>par</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2(+z2)</td>
<td>par</td>
<td>seq</td>
<td>par</td>
<td>seq</td>
<td>par</td>
<td></td>
</tr>
<tr>
<td>t3(-z3)</td>
<td>seq</td>
<td>par</td>
<td>seq</td>
<td>par</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4(-z4)</td>
<td>seq</td>
<td>seq</td>
<td>seq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5(-z1)</td>
<td>seq</td>
<td>par</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t6(+z3)</td>
<td>par</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t7(-z4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3. Triangular matrix defining relations seq and par on the set of labelled transitions

\[
<\text{composite\_operator}>
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{BEGIN} \quad <\text{sequence\_of\_operators}> \quad \text{END} \\
<\text{operator\_1}> \quad ; \quad <\text{sequence\_of\_operators}> \\
<\text{operator\_2}> \quad ; \quad <\text{sequence\_of\_operators}> \\
\end{array}
\end{array}
\]

...........

<\text{operator\_-K}>

(a) initial structure

\[
<\text{composite\_operator} \quad \text{BEGIN} \ldots \text{END}> \\
<\text{operator\_1}> \quad <\text{operator\_2}> \quad \ldots \quad <\text{operator\_K}>
\]

(b) reduced structure

Fig. 4. Illustration of parse tree reduction
Fig. 5. Example of specification fragment tree

Fig. 6. A variant of structural organisation of Prolog prototype software implementation for specification validation environment