ESPRIT Project 2072 ECIP2

European CAD Integration Project

Prepared by

Cristian A. Giumale (University of Manchester),

An Information Model of VHDL
Report

ECIP2/UM/015-1 **01-05-93**

This report addresses the information modelling of both structural and behavioural aspects of VHDL. It superseeds the Report ECIP2/UM/008-1 "Towards A VHDL Model".

Contents

1 Introduction

VHDL is intended for use as a standard hardware design language in the microelectronic design community. It describes general digital hardware devices at an abstract level. The description is performed at the behavioural, timing and structural levels. An abstract event-driven simulation model is used to define the semantics of the language constructs by specifying the way they are evaluated while the simulation process unfolds.

There are many attempts to describe the VHDL semantics, particularly its behavioural semantics. The main purpose of these efforts is to capture in a clear and unambiguous notation what is described informatively in the VHDL reference manual and then to use the formal descriptions in order prove properties of VHDL programs or to help with the implementation of related software tools, especially simulators.

The methods used for modelling VHDL can be classified into formal, ranging from predicate calculus [VT92], Milner's CCS calculus and temporal logic [SMW92] to Petri nets and data oriented, mainly information modelling [MR92] [WD92] or other adhoc methods [SO93]. However, the existing models are only partial. Usually the formal models address the simplified behavioural semantics of VHDL, while the information models tackle only part of the structural aspects of the language. Moreover, there is no model so far, to my knowledge, which tackles the elaboration phase of VHDL.

Although the various apects of VHDL can be treated separately, they are strongly interrelated and a comprehensive model of VHDL can neglect neither of them. For example, the behaviour of a block or of an instantiated component cannot be described without sensible insight into the structural and elaboration aspects of VHDL. An early attempt to join structure and behaviour into a single information model [GC92-2] was also too superficial and, in addition, it completely neglected the VHDL elaboration phase.

This report is an attempt to produce an almost complete **core model** of VHDL from the structural, elaboration, behavioural and execution points of view. In addition, it models VHDL typing and configuration. Elaboration cannot be achieved without configuring a VHDL description, and important structural and behavioural aspects cannot be modelled properly without describing, at least superficially, the VHDL typing schema. However, the VHDL computational aspects such as procedures, variables, expressions, which would clutter the model with unnecessary information, are avoided.

The purpose of the model is to describe in a more formal way the salient features of VHDL. It could be used finally as a basis to check conformance of VHDL oriented software, for example simulators, against the VHDL standard.

The present model of VHDL is structured into several layers. The structuring criteria is twofold. First, layers correspond to the modelled aspect of VHDL. For instance, there are structure, type and behaviour layers. Second, layers correspond to the abstraction level of modelling. For example, there is a generic block layer, describing concepts such as program block nesting and visibility in general, and a time layer, describing what is meant by event and specific temporal relationships between events.

The model is almost complete. Signal disconnection and generate statements have not been modelled. While the generate statement can be left apart, signal disconnection is nevertheless an important behavioural aspect of VHDL and should be integrated within a more complete, future model.

The model tries to avoid as much as possible the use of functions, in order to describe complex constraints. However, there are cases, some of them quite simple, particularly for specifiying logical implications, when functions could not be avoided. In addition, there are points in the model where the clarity is blured further not as much because of the complexity of the modelled aspect but merely due to lack of suitable mechanisms in EXPRESS.

2 General Modelling Decisions and Style

As in programming, a model can be developed adhering to a specific "modelling style". The style used here relies on layering and specialization.

The model is divided in several schemas each corresponding to a specific aspect of VHDL. These schemas form a hierarchy which follows the interrelationships and, in fact, the 'natural' hierarchy, of the modelled aspects of VHDL. Generally, the concepts described on the layer `i` of the hierarchy rely solely on concepts described on lower layers `j <= i`, although there are few exceptions to this rule.

For example, the behaviour schema describes the entity `signal wait transaction` which is the result of executing a signal wait statment from a VHDL process. The entities `signal wait statement` and `process` are however structural aspects of VHDL, being modelled in the structure schema, at a lower level in the hierarchy.

Another example is that of the entities `architecture body` and `block` described in the `structure schema`. Both rely on the entity `generic block` which is one level lower, in the generic block schema.

Nevertheless, there is a major difference between the two examples above. From the conceptual point of view it cannot be said that a `signal wait transaction` is a sort of `signal wait statement`. Therefore, the entity `signal wait transaction` can hardly be seen as a subtype of the `signal wait statement` entity. The right thing to say is that a signal wait transaction `SW-T` has a parent (is generated by) which is a specific signal wait statement `SW`.

On the other hand, both the `architecture body` and the `block` are a sort of `generic block`. They are specializations of the entity `generic block` being related to it by **isa** conceptual relationship.

The above observation shows that entities in layer `i` of the model can be specializations of entities from layers `j, j <= i` or they can use, eventually in the role of 'parent', entities from higher layers. Such a relationship can extend between all entities of two different layers thus making a model layer `i` a specialization of a layer `j, j < i`. This is the case of the design unit schema which contains mainly specializations of the entities from the configuration schema and structure schema, as suggested in figure 1.

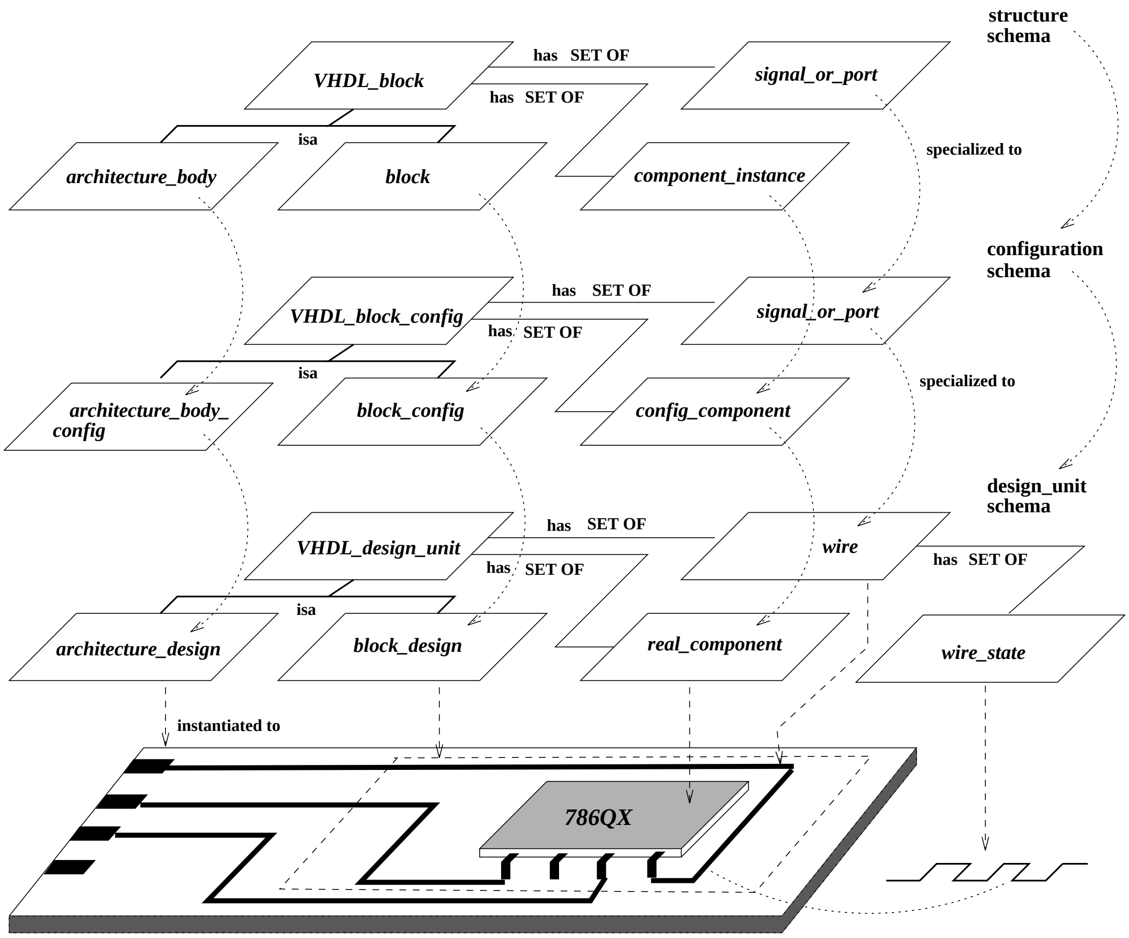

Figure 1: Layer specialization

Layering a model and studying possible relationships between model layers could well be a subject of research, similar to the program structure field in conventional programming. This could lead to important extensions to the modelling languages, particularly EXPRESS, in order to cope with schema level operations (composition, specialization inheritance etc.)

The layers of the VHDL model are discussed in section 4. Nevertheless, the problem of entity subtyping and specialization deserves further insight here, since its use is double fold.

First, the subtyping is used in a simple way to construct hierarchies of entities which share common features. For example, if `E1, E2` are entities with a common attribute `A` of type `T` which satisfies a constraint `R(A)` then the EXPRESS specification of these entities is as shown below.

```
                ENTITY E ABSTRACT SUPERTYPE OF(ONEOF(E1,E2));
                   A: T;
                WHERE R(A);
                END_ENTITY;

ENTITY E1 SUBTYPE OF(E);     ENTITY E2 SUBTYPE OF(E);
  A1: T1;                       A2: T2;     -- Additional attributes
WHERE R1(A1);                WHERE R2(A2);  -- Additional rules
END_ENTITY;                  END_ENTITY;
```

A typical example is that of the `block` and `architecture body` entities from the VHDL model. They have common attributes such as `local objects` and `concurrent statements`. Both are subtypes of the `VHDL block` entity.

Second, the subtyping combined with specialization is intended to implement the mathematical notion of 'morphism'. Consider, for example, the case of an entity `E` which is specialized to an entity `E'`.

```
ENTITY E SUPERTYPE OF(E');     ENTITY E' SUBTYPE OF(E);
  A: T;                          SELF\E.A: T';
DERIVE                         DERIVE
  D: TD:= F(A);                  D': TD':= F'(A);
INVERSE                        INVERSE
  I: TI FOR B;                   I': TI' FOR B;
WHERE                          WHERE
  R(A);                          D'\E = SELF\E.D;
END_ENTITY;                    END_ENTITY;

ENTITY TI SUPERTYPE OF(TI');   ENTITY TI' SUBTYPE OF(TI);
  B: E;                          SELF\TI.B: E';
  W: T;
END_ENTITY;                    END_ENTITY;
```

The entity `E'` is obtained from `E` by

- Making E' a subtype of E;
- Making the type T' a subtype of T, TD' a subtype of TD and TI' a subtype of TI.
- Specializing the type of the attribute A and defining new attributes D', I' as specializations of D, I;
- Eventually adding constraints to make sure that D', I' are correct specializations of D, I.

These transformations and some additional constraints make the morphism diagrams from figure 2 commute. The diagram (a) shows that a constraint R inherited from entity E remains valid in the context of entity E'. The diagram (b) addresses the problem of derived specialized attributes, such as D' of E', whereas the diagram (c) treats the case of inverse attributes. The notation x|t, where t is a type and x is an instance of a subtype t' of t stands for the type conversion from t' to t. The conversion is considered pervasive (i.e. if x is a structured value then the conversion applies to each element of the structure).

Therefore, if an entity E has an explicit attribute A of type T then E can be specialized to an entity E' with an attribute A of type T' iff T' is a specialization of T. This rule can be found in the EXPRESS reference manual.

In the case of derived attributes, if F: T -> TD is a (deriving) function used by E and F': T' ->TD' is its specialization used by E', then TD' must be a subtype of TD and the relation x = x'|T <=> F(x)|TD = F'(x')|TD must hold for all x IN T, x' IN T'.

For inverse attributes, such as I within the entity E, the specialization mechanism, in the variant shown above, is more complex. First, a local (new) inverse attribute I' is declared. Second, the constraints obeyed by I are eventually reformulated for I'.

Another variant of specializing the inverse attributes is to replace them by direct attibutes and additional constraints which enforce the inverse relationship and then to specialize the direct attributes. This variant is simpler and more clear than the one above. For example, the signal_or_port source entity has an attribute called target which normally should be declared target: INVERSE signal_or_port FOR sources. Instead target is declared as a direct attribute, the inverse relationship being enforced by a specific rule within the signal_or_port entity. In general, the problem of inverse attributes within entities which has to be specialized is solved as follows.

```
ENTITY E SUPERTYPE OF(E');          ENTITY E' SUBTYPE OF(E);
  I: TI;                              SELF\E.I: TI';
WHERE I.B :=: SELF;                 END_ENTITY;
END_ENTITY;

ENTITY TI SUPERTYPE OF(TI');        ENTITY TI' SUBTYPE OF(TI);
  B: E;                               SELF\TI.B: E';
WHERE B.I :=: SELF;                 END_ENTITY;
END_ENTITY;
```

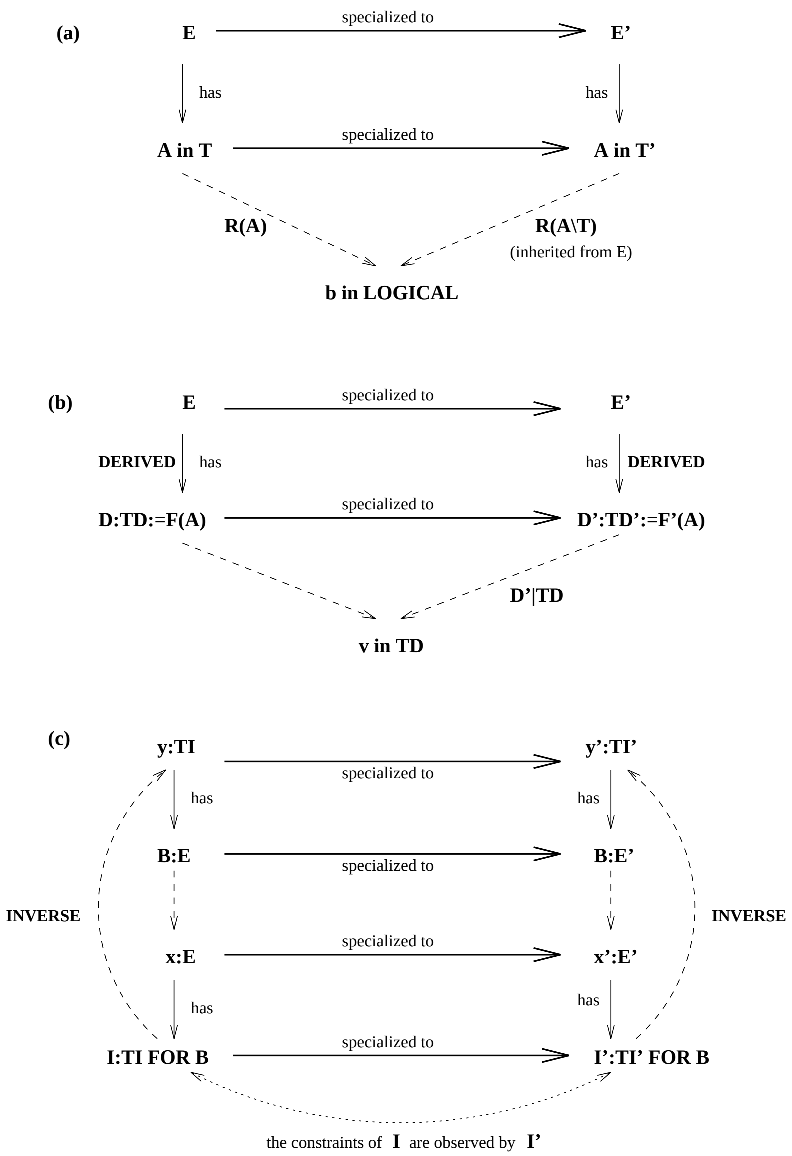

Figure 2: Specialization of entities

Obviously, the entity specializations above amount to an algebra morphism. From this point of view EXPRESS is not so well equipped, or at least its semantics is not clear enough. An interesting direction of research would be to put more power into the specialization mechanism of EXPRESS. After all, the specialization could be done not only for particular entities but also at the level of an entire schema, as suggested in figure 1. That would be very useful indeed.

Nevertheless, VHDL model uses extensively the specialization mechanism. All the entities in the design unit schema, including the `architecture design` which is an executable (or simulatable) unit, are specializations of the entities from the configuration schema and structure schema. Signals from the structure layer are specialised to 'wires' which keep their behavioural history in the form of a set of states over the entire simulation process life-time. Processes become real processes which store the history of their wait and assign transactions etc. The specialization mechanism preserves all the structural properties while transforming a design pattern into a 'real' design where some entities at the structural and configuration level generate a set of real objects characterised by states.

3 VHDL Modelling Space

In VHDL, the status of a device depends on the states of its signals. In turn, the status of a signal can be modified by concurrent processes local to the device. A process can be blocked, waiting for some time delay to be consumed or for some change in the states of a set of signals, or active thus modifying the signals of the device. The VHDL sample below specifies a very simple process which describes an oscillator. The signal `s` is complemented every 20 nano-seconds. After each signal assignment `s <= (not s) after 20ns` the process `P` is blocked for 20ns waiting for `s` to change its value.

```
entity oscillator is
      port (out1: bit);
      constant delay: Time:= 20 ns;
end entity_oscillator;

architecture oscillator_behaviour of oscillator is
      signal s: bit:= '0';
begin
      P: process
         begin
            s <= (not s) after delay;
            wait on s;
         end process_P;

      out1 <= s;
end architecture_oscillator;
```

To model the behaviour of process `P` and, in general, to model the time-dependent behavioural semantics of VHDL the following kinds of concept are necessary:

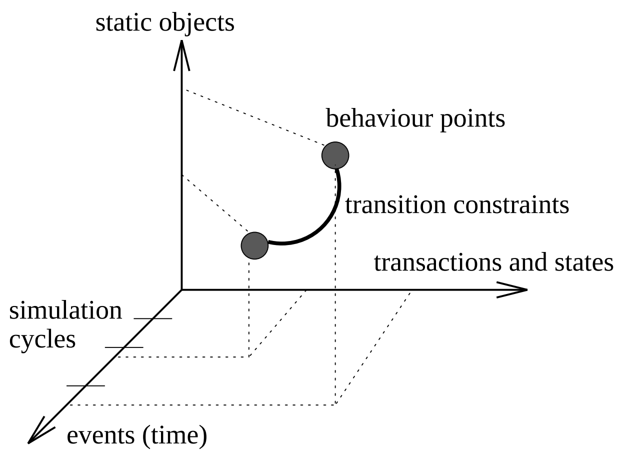

Figure 3: The VHDL modelling space

- Concepts which stand for static (time independent) VHDL objects such as component, port, signal, process, signal assignment. Only the state of such an object can change in time, not the object itself.
- Concepts which designate VHDL dynamic (time dependent) objects. These objects are abstract, such as process execution for example, and are divided into transactions and states. A transaction can be looked at as representing the effect of an "executable" VHDL static object (e.g `signal assignment`.) A state represents the changing part of a VHDL static object. Therefore we can speak of, for example, assign transactions and signal states.

At each moment in time a static object of the model can be associated with different dynamic objects. Therefore a dynamic object plays its role only at a given moment in time. Static objects, dynamic objects (transactions and states), and time events are the axes of the three dimensional VHDL modelling space, as illustrated in figure 3. The behaviour of a VHDL design is a surface on this space. Modelling VHDL behavioural semantics requires the specification of relationships between points along the three axes so that a transition from one point on a generic behaviour surface to another point is a valid one.

The relationships along the structure axis describe what is meant by a correct structural design. For example, the signal `s` used within the process `P` of the architecture body `oscilator behaviour` must be 'visible' from the point of its use (i.e. declared in a lexical environment including the process `P`).

The relationships along time (event) axis describe temporal and cause-effect relationships between the transactions and states used by the model.

There are many time dependent relationships to model in VHDL. An example is the `cause - effect` dependency between the wait transactions generated by processes belonging to a real component (an elaborated component instance which has been already configured). A wait transaction is generated by executing a wait statement from within a process, in order to wait for some signals, as `wait for s` in process `P` above, or for a fixed time delay. A wait transaction is the effect of fulfilling a prior wait transaction from the same process. For example the wait transaction `wait for s` from the simulation cycle `i > 0` is the cause of the wait transaction `wait for s` in cycle `i+1`.

VHDL modelling is based on the concept of `simulation cycle`. A cycle starts when processes become unblocked, i.e when some wait transactions are satisfied (the goal waited for is satisfied), for example when the signals waited for changed their values. A cycle ends when the unblocked processes become blocked and, therefore, after new signal assignments are evaluated and wait transactions are generated. Since null delays are valid in signal assignments, processes waiting for such signals can complete instantly and, therefore, there can be successive cycles with the same time value. The order of these cycles is correctly captured by the time model used [GCKH93].

A flow of successive simulation cycles, for a given elaborated design, is clustered in a 'simulation process'. The `simulation process` is taken as a base to specify constraints between transactions and stated in different simulation cycles which, in turn, are considered as specific events. Grouping transactions and states within simulation cycles and considering a simulation cycle as an event, is the main mechanism to specify temporal constraints over the domain of dynamic VHDL concepts.

4 VHDL Modelling Layers

The VHDL model spans across several layers starting from the generic block structure and VHDL data types and ending at the time model layer. The actual behaviour model is in between the elaboration layer (the design unit schema) and the time layer and encompasses the behaviour and simulation process layers as shown in figure 4.

It is interesting to note that the generic block, type and time modelling layers act as ordering referentials for the entities at the intermediate leayers of the model. For instance, in a VHDL architecture or block, ports and signals are connected observing type and block visibility constraints. Similarly, their states and the transactions which generated these states are grouped in simulation cycles which, in turn, are considered events ordered in time.

Thinking of the model in terms of layers helps not only to control inherent complexity but, it structures the way concepts are represented thus avoiding the duplication of already described aspects. For example, structural aspects described in the structure schema are not repeated at the level of library schema, design unit schema and behaviour schema. This is possible due to the specialization mechanism used in the modelling process. Each modelling layer focuses a specific aspect of VHDL. Therefore the layer describes only those entities and constraints related to the modelled aspect. Here is a short presentation of the modelling layers used in the present VHDL model.

The **generic block schema** is a general model of what is meant by a block in the programming domain. The model of a `generic block` and `generic object` describes the concepts of block nesting, block object locality and block object visibility.

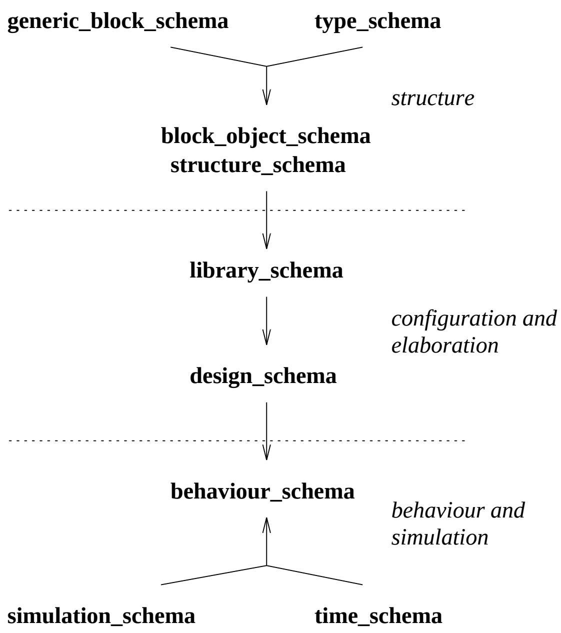

Figure 4: Layers of VHDL modelling

The **type schema** represents a model of the VHDL types and values. It is not complete for an obvious reason: a type, in general, is characterised not only by its set of values but, in addition, by a set of operators and the semantics of these operators. To model fully a data type means to model the semantics of the type operators. This is a subject of research by itself which could be addressed as modelling Abstract Data Types. The model given here does not describe the data type operators. However, the model contains the necessary information required at the higher levels of the VHDL model.

Why is it necessary to model VHDL types ? If the VHDL model describes guarded (concurrent) assignments then there are constraints which say that "the effect of an assignment if enabled if the value of its guarded signal is `VHDL true`". The value `VHDL true` however is a constant out of the set of values which characterises the VHDL `boolean type` which, in turn, is a predefined subtype of the VHDL `enumeration type`. Another reason, is given by the must of writing constraints which address the type compatibility. For example, the type of a source (driver) of a resolved signal must be compatible with the type of the resolved signal. A connection between signals might as well involve type conversion (done by a `conversion function`.

A typing schema cannot be "closed" in the sense that it fully describes itself. Some external `basic` types must be used in the description. From this point of view the `type schema` explicits the `VHDL -> EXPRESS` type morphism.

The **block object schema.** At this level the main objects of VHDL are described: signals, ports, generic constants, components, resolution and type conversion functions. These are named, and some of them typed, block objects defined as specializations of the `generic object`. They will be clustered within architecture bodies and blocks at the structure schema level.

The most important attribute of a signal or port is the set of its sources, which can be port connections and signal assignments. A `port connection` is used by a `component instance` to describe how the component ports are connected to other signals. A `signal assignment` is used within a `process` to assign a computed value to a signal.

The **structure schema** describes the contents and structure of a `VHDL block`, seen as the specialization of the `generic block`. The capability of inheriting objects from packages is also modelled. From this point of view the attribute `local objects` from a `VHDL block` represent not only the objects declared within the block (local objects) but, in addition, the objects inherited from the used packages and from an `entity declaration`, in the case of an `architecture body`. In addition, this level models the component instances, processes, concurrent signal assignments and the elements they use: port maps, wait statements and sequential signal assignments.

The **library schema** focuses on modelling the VHDL library units, including the configuration declaration, used to configure architectures and blocks. Therefore, the model addresses the problem of particularizing the values of the generic constants and that of configuring the component instances from an architecture or a block. The main thrust is towards specializing the hierarchy of the `VHDL block` entity in order to describe what is meant by a configured block and a configured architecture. The entities `architecture config` and `block config` serve this task. It can be said that the library schema corresponds to the configuration phase of a VHDL description.

The **design unit schema** corresponds to the elaboration phase of a VHDL description. The main idea used at this level is to specialize the entities from the library schema and structure schema turning them into isomorph entities able to store their own behaviour history seen as a set of states over the simulation process, a state per simulation cycle. The strong point is that entities are not specialized separately. Instead the specialization addresses full hierarchies of entities.

This layer can be seen in its entirety as a specialization of the library schema and structure schema layers. In particular, signals and ports are specialized to `wires` which are meant to stand for real elements of a design. Each wire stores its own `wire history` (set of wire states) over the simulation process. Processes are specialized to `real processes` which work with wires by means of `wire assignment`, itself the specialization of `signal assignment`. All these objects are grouped in an `architecture design` which, in addition, contains `block designs` and `real components`. The structural properties are inherited from the structure and library schemas. An instance of an `architecture design` is equivalent to a chip or a board containing the complete history of its simulation.

The **behaviour schema** models the behavioural aspects of VHDL. In particular it focusses on temporal cause-effect relationships between `transactions` and `wire states`. As discussed in section 3 a transaction is an entity which describes the executing phase of a VHDL executable "static" object such as, let say, a `signal wait` or a `wire assignment`. A state describes the current status of `wire`. The set of transactions form the history of the `real process` which generated them. A set of `wire states` form the history of the `wire` they belong to. Each transaction and state is associated to a `simulation cycle` which is a subtype of `event`. Therefore, considering the histories of processes and wires, constraints can be written to capture the relevant cause-effect relationships.

The **simulation schema** describes the flow of simulation. It tackles the problem of linking together the `cause-effects` modelled at the behaviour schema level. It also models singular points during the simulation process: the start and the end of simulation. The main entity at this level is the `simulation cycle`. It clusters all the transactions and wire states corresponding to the point in time represented by the `simulation cycle`. The anchoring in time is achieved by making the `simulation cycle` a subtype of `event`.

The **time schema** is a generic model of time, which describes generic events and their temporal relationships. The model is taken from [GC92-1] and can cope with instantaneous events. Implicitly it can cope with simulation cycles which are successive as far as the simulation process is concerned but can have the same time values. This phenomenon happens when using "delta delays".

5 Modelling the VHDL Structure

The main structural entities used by the structural model of VHDL are defined in the block object schema. They are signals, ports, generic constants, components, resolution functions and type conversion functions. All of them are named objects and, apart from `component`, typed objects. The general subtyping diagram for the main entities defined within the structure schema is illustrated in figure 5.

An important entity, which is associated with a `signal_or_port` is the `signal_or_port source` which can be a `signal assignment` from a `process` or a `port connection` from a `component instance`. A signal can have multiple sources and therefore can be resolved. For each signal source there will be a corresponding signal `driver` at the behavioural modelling level. A source contains only structural information while the driver contains a waveform anchored to a specific simulation cycle.

Note that all entities which are subtypes of `atomic object` act as templates. An instance of such an object can be member of multiple instances of the entities which describe the VHDL architecture and block. This view of an `atomic object` underlines the difference between an object identifier and the object identified within a program. The same identifier can stand for different objects, eventually of the same type, which exist only when the program is executed.

The **entity declaration** corresponds to the VHDL entity. An `entity declaration` can declare signals, ports, types, subprograms (resolution functions and type conversion functions used in port maps) and generic constants. It can also inherit similar objects from 'used' packages.

The **architecture body** corresponds to the VHDL architecture and can declare signals, types, components and subprograms. It can inherit similar objects from 'used' packages. In addition, an `architecture body` can contain instances of the entity `concurrent statement`. An outstanding optional guard signal is singled out in an `architecture body`. This signal is used by the guarded concurrent assignments.

The **block** entity represents a VHDL block. A `block` can declare ports, signals and components. It can use packages and contain concurrent statements. The `block` is itself a `concurrent`

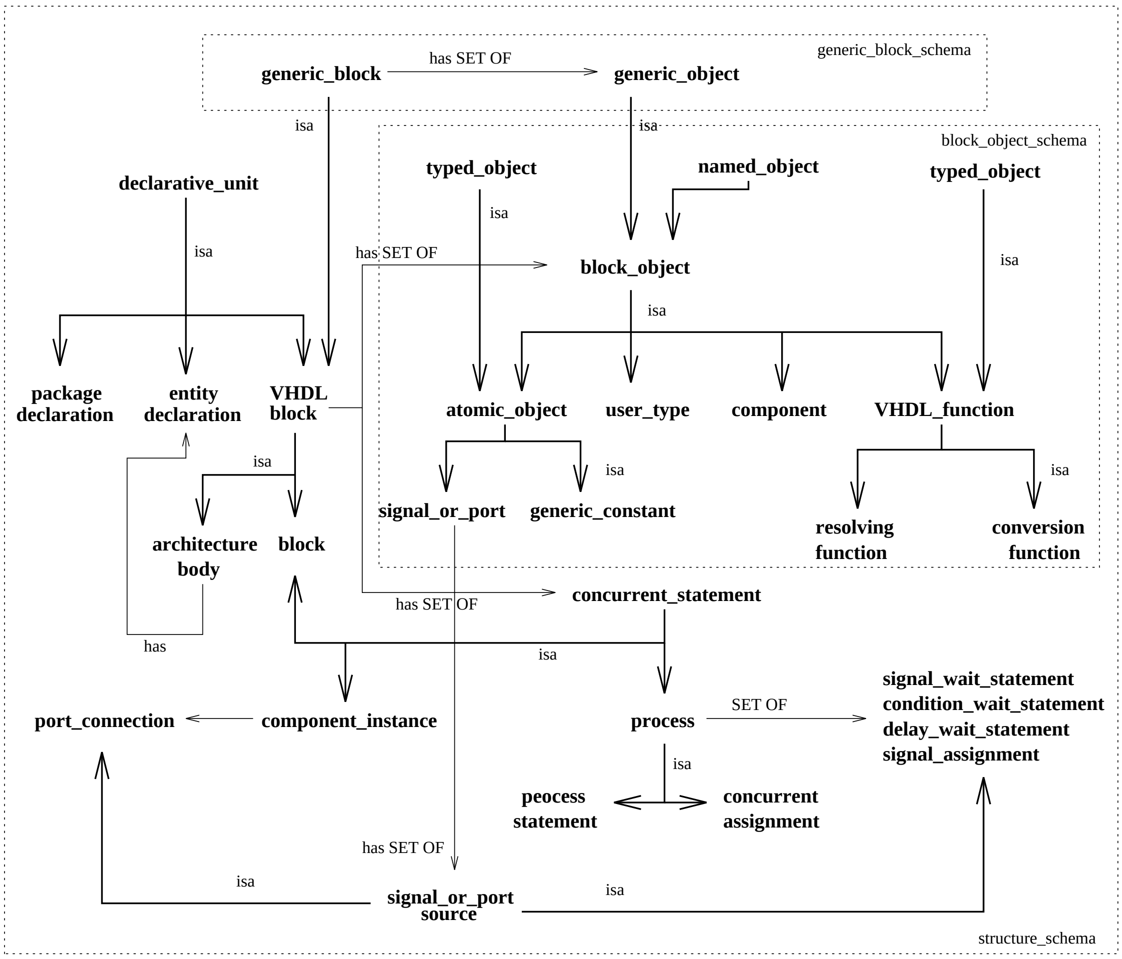

Figure 5: The structure modelling level

`statement`. Similarly to an `architecture body` a `block` can declare a guard signal. Note that the implicitly guarded block has always an equivalent block which uses an explicitly declared guard signal.

The **package declaration** is seen as a cluster of declarations of types, signals, subprograms, constants and components. A package declaration can be used by another `package declaration` and by any of `entity declaration, block` and `architecture body` in order to inherit the declared objects.

All of the above entities although different, have a common feature: they can be regarded as 'declarators' of atomic objects: types, signals, ports, constants and components. From this point of view they all are subtypes of the `declarative unit` entity. The use of the `declarative unit` entity avoids the repetition of common attributes and constraints within each of the 'declarator-type' entity. Moreover, a `declarative unit` models the mechanism of inheriting atomic object declarations from used packages. These inherited objects are considered part of the atomic objects declared locally, within the `declarative unit`.

However, the entities `block` and `architecture body` are block structured compounds and, therefore, they are also subtypes of the `generic block`, provided that the local objects of the parent `generic block` correspond to the objects declared within the parent `declarative unit`. Moreover, a `block` is a subtype of `concurrent statement`.

Apart from the entities mentioned above, the structure layer model describes 'executable' VHDL statements: concurrent and sequential statements. The entity `concurrent statement` stands for `block`, `component instance` and `process`.

The **component instance** represents a VHDL component instantiation statement. It contains a port map and a generic constant map. The `generic constant map` is used to redefine some generic constants of the instantiated component. The `port map` is a set of port connections.

A `port connection` describes the way a port of the instantiated component is connected to a signal or port. A `port connection` has two main attributes which can be seen from two different perspectives. First, as a pair (`locall`: port, `actual`: signal or port). (Since `local` is a keyword in EXPRESS, `locall` is used instead). The `locall` is a port of the instantiated component, the `actual` is a signal or a port of the `entity declaration` associated with the surrounding `architecture body`. Second, as a pair (`target: signal or port`, `source: signal or port`). From the `port connection` point of view the precise mapping `[locall,actual] -> [source,target]` cannot be DERIVED. The information conveyed solely by the VHDL port map is not complete for deciding which is the target and which is the source of the connection. For example, consider the case of `inout` ports. Therefore what the model states is the condition of a correct mapping, i.e. the mapping `[locall,actual] -> [source,target]` must be one to one.

The **process** entity represents both the VHDL process statement and the concurrent signal assignment statement. It contains (sequential) signal assignments and wait statements: signal wait statements, delay wait statements and condition wait statements. The signals used by the `process` are grouped into: `controlled signals` - appearing in the left hand side of assignments, `sensitivity signals` - contained within signal wait statements, `condition signals`

- used within the conditions from the condition wait statements and **assign signals** - used in the right hand side of the process signal assignments. These signals must be visible (i.e. declared within the surrounding VHDL block or inherited via use of packages.) Note that the computational aspects of a process are not modelled. They are not relevant from the point of view of the behavioural semantics of VHDL. From this point of view the use of variables and subprograms within packages and blocks has not been considered. There is one notable exception: the resolution and type conversion functions which play important roles at the level of the VHDL behavioural semantics.

The **assignment statement** entity represents all the assignments with the same signal as target from a `process`. When the process is executed this 'synthetic' assignment produces as effect a list of assigned values. The order of these values within the list corresponds to the order of executing signal assignments as part of the overall execution of the representing 'synthetic' assignment.

As an example which shows the meaning of the 'executable' entities discussed so far, consider the process below.

```
              P: process
                 begin
                   s <= (not s) after delay; <---|
signal_wait ---->  wait on s;                    |
                   if x then s <= '1' after 1; <-|
                        else s <= '0' after 1; <-| synthetic assignment
                   end if;                       |
delay_wait  ---->  wait for 3 ns;                |    s <= f(s,x)
                   s <= '1'; <-------------------|
condition_wait ->  wait until x;
                 end process_P;


sensitivity_signals(P)= [s]   condition_signals(P)= [x]
controlled_signals(P)=  [s]   consulted_signals(P)= [s,x]
assign_signals(P)=      [s]
```

The `concurrent signal assignment` is a subtype of `process`. This acknowledges the fact that a concurrent assignment is equivalent to a process as follows.

```
s <= not a;          <=>  P: process  -- normal process
                          begin
                            s <= not a;
                            wait on a;
                          end;


s <= guarded not a;  <=>  P: process  -- guarded process
                          begin
```

```
                if GUARD then s <= not a;
                         else s <= null;
                         -- if s is a guarded signal
                end if;
                wait on GUARD;
              end;

assignments(P) = [s <= not a]
signal_waits(P) = [wait on GUARD]
delay_waits(P) = condition_waits(P) = []
```

Note that the entity `process` has an additional attribute `is guarded`, which can be true only if the process is a parent of a guarded concurrent assignment. A normal `process statement` cannot be guarded. The `is guarded` attribute is artificial, and helps with the modelling of the `concurrent statement` as a particular `process`.

6 Modelling the VHDL Design Configuration and Elaboration

The configuration of a VHDL design is based on particularising the component instances and the generic constants contained within the design `architecture body`. Each component instance is associated to a pair (`entity declaration, architecture body`).

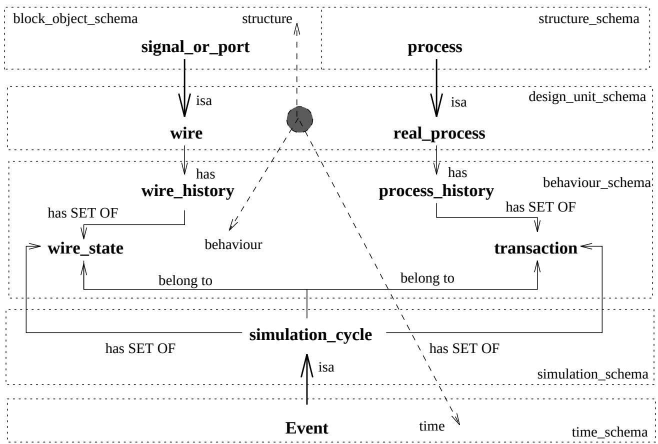

Figure 6: The elaboration modelling level

The elaboration of a design consists in recursively instantiating the `architecture body` of the design by replacing the component instances it contains with instantiated architecture designs as specified in the configuration phase. In the process the block objects of architecture bodies and blocks are treated as instances rather than templates. Signals and ports are turned into wires, processes are substituted by real processes and the component instances by real components. The modelling of all of these relies heavily on entity specialization as discussed in section 2.

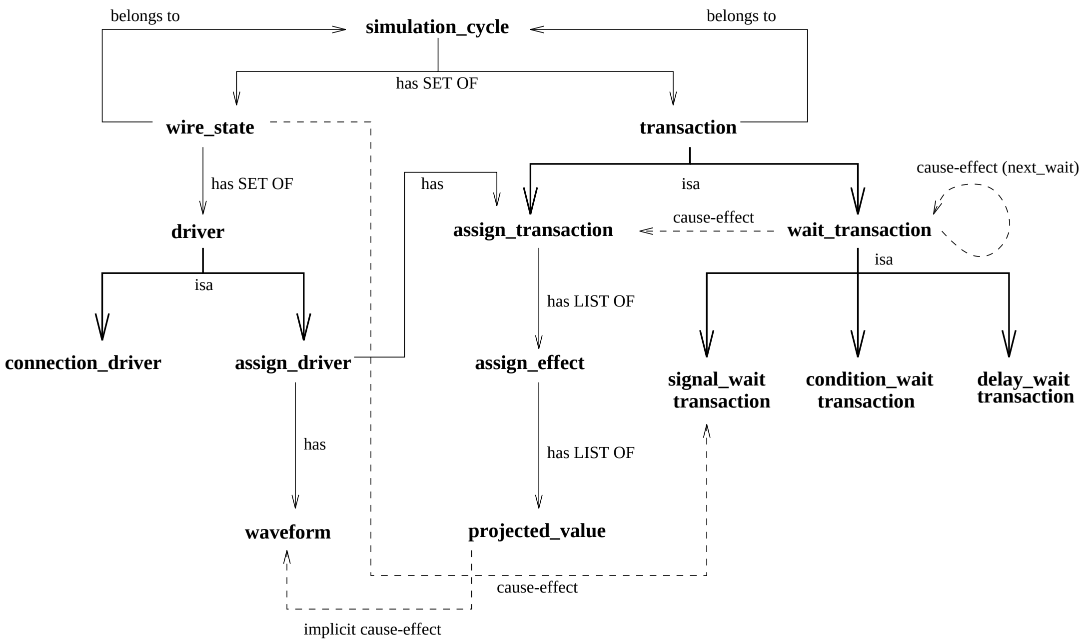

Figure 7: Cause-Effect behavioural relationships

The main entities at this level of modelling are `wire` and `real process`. Both have an attribute called `behaviour`. The wire behaviour is the history of wire states over the entire simulation process of the wire parent design. The real process behaviour is the history of the transactions (assignments and waits) generated by executing the process during the simulation process. From this perspective it can be said that this level of modelling is the point where the three axes of the modelling space converge: the structure axis, the behaviour axis and the time axis as suggested in figure 6.

7 Modelling the VHDL Behaviour

The entities at this level correspond to states and transactions as discussed in section 3. The general subtyping diagram for the main entities defined within the behaviour schema is illustrated in figure 7. The relationships between these entities show how the simulation process unfolds, from one cycle to another.

7.1 Transactions and wire states

There are three main behavioural entities: wire state, wait transaction and assign transaction.

The **wire state** corresponds to the state of a wire in a given simulation cycle. For each wire of the simulated design the states of a wire over the entire simulation process are grouped within the `wire history`. A `wire state` contains the set of `drivers`, one for each source of the wire (assignment or connection), the associated `resolution computation`, the current value and the state the wire had in the previous simulation cycle (except for the beginning of the simulation process). The previous state is necessary to decide weather there is an event on the wire, i.e. if there has been a change in the value of the wire since the last simulation cycle.

The **wait transaction** entity corresponds to the execution of a wait statement. The subtypes of a `wait transaction` are: `signal wait transaction`, `condition wait transaction` and `delay wait transaction`. Transactions corresponding to the same `real process` are grouped within the `real process history`. As a difference to the wire states, the wait transactions are spread non uniformly across the simulation process. A process might not have wait transactions in each simulation cycle.

The **assign transaction** entity corresponds to the execution of a 'synthetic assignment' from within a `real process`. Note that the process can represent an equivalent concurrent assignment. The main attribute of an `assign transaction` is the list of assign effects. An `assign effect` models the effect of executing a particular assignment represented by the synthetic `signal assignment`. An `assign effect` itself can inject several values into the `waveform` of the `assignment driver` associated to the wire which is the target of the `wire assignment` which generated the `assign transaction` as illustrated in figure 8.

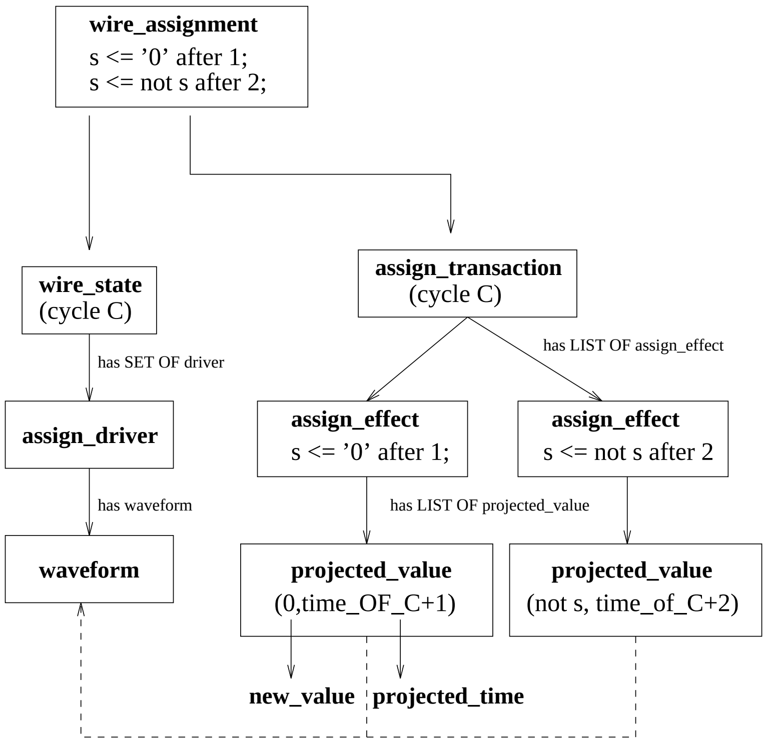

Figure 8: The anatomy of an assign transaction

Since there can be multiple assignments to the same wire while a `real process` is active LIST instead of SET of `assign effect` is used to model the order in which these assignments are executed.

Apart from the above main entities the following make the picture of the behavioural modelling level more complete.

The **driver** entity is the equivalent of the `signal or port source` from the structure schema. It is either an `assign driver`, corresponding to the driving of a signal by a 'synthetic assignment' contained within a `process`, or a `connection driver`, corresponding to the driving of a signal by force of a component instance port connection. Both type of drivers have in common the following attributes: `driver current value, sim cycle, parent wire state` and `parent wire source`.

The `sim cycle` is the simulation cycle of the `parent wire state` and the `parent wire source` is the corresponding `wire source` which inherits the structural properties of a `signal or port source`. However, the `driver current value` is derived differently for the two types of drivers.

The `driver current value` of an `assign driver` is computed on the base of the driver `current waveform`, updated according to the projected values of the associated assign transaction. In order to be able to perform this computation the status the driver had in the previous simulation cycle, if any, is explicitly kept as the value of the attribute `previous driver`. The `current waveform` is the `new waveform` of the `previous driver`, if this exists, or the 'initial waveform' containing the default value of the driven wire, if the simulation just started.

The computation of the `driver current value` takes also into account the derived value of the attribute `assign inhibit` which is used to model the behaviour of guarded concurrent assignments and the behaviour of a blocked process. If the `wire assignment` the driver is associated with is contained within a guarded process (i.e. it is a guarded concurrent assignment) then the assignment effect must be discarded if the guard is `VHDL false`. Similarly, if the process of the `wire assignment` is blocked then the corresponding `current assign` transaction does not exist (according to the `wait transaction - wait transaction` behavioural relationship, as discussed in section 7.2) and therefore the assignment has no effect on the driver waveform.

In the case of a `connection driver` the `driver current value` is the value of the connection source wire in the same simulation cycle. Note that the value of the source wire of a component port connection is equal at any time with the value of the wire corresponding to the entity port used to configurate and then elaborate the design (see the constraint from the entity `component port wire connection` in the `design unit schema`).

The current value of a wire state can be computed according to a `resolution computation` if the parent wire is resolved. Similarly, the value of a `connection driver` can be type converted by a `conversion computation`.

The **resolution computation, type conversion computation** entities stand for the current execution of a resolution function or a type conversion function. How the computing process goes on is not important. What matters is the value which is produced by these computations, in the case they exist.

7.2 Behavioural relationships

The transactions and the wire states are related by important cause - effect relationships which govern the flow of the simulation process.

The relationship **wire state - signal wait transaction**.

It is important to note that a `wire state` which has changed its value can have several effects. Otherwise said, a `wire state` can be considered as the cause of satisfying several signal wait transactions. Consider the example below.

```
    cycle Ci                                   cycle Cj (is_after(Cj,Ci))
----------------------------------------------------------------------

P: process             execution         P: process
   begin                   |                begin                   <__
     s <= not s after 3;  V                   s <= not s after 3;     |
     wait on s;          ___                     wait on s;       ___ |
   end;                                        end;                 |  |
                                                                    V__|
----------------------------------------------------------------------


  signal_wait_transaction   --------->   wire_state WSj
  WTi (wait on s)           P blocked      wire: s
                                           signal_event= TRUE
```

Assume the simulation is in cycle `Ci` and the value of `s` is 0. The assignment `s <= not s after 3` is executed. It produces the projected value `(1 after 3)` to be added to the waveform contained by the unique driver of `s`. The next statement to be executed in simulation cycle `Ci` is `wait for s`, which generates a signal wait transaction `WTj`. The `wait transaction` will be satisfied when `s` will change its value.

Assume that simulation progressed and the current cycle `Cj` is that corresponding to the change of the value of `s`. The wire state `WSj`, corresponding to `s`, records the change of `s` and also records the fact that the signal wait transaction `WTi` is satisfied. Therefore, the wire state `WSj` can be considered as the cause of satisfying the wait transaction `WTi`. The following rule states this relationship showing that in the simulation interval from `Ci` to `Cj` all wire states for `s` have their `signal event` attribute FALSE (there is no change in the value of `s`).

```
RULE nearest_signal_wait_outcome FOR (signal_wait_transaction, wire_state);
-- Between a signal_wait and its outcome_cycle, if this exists,
-- there is no wire_state such that the parent_wire is in the
-- sensitivity_signals of the signal_wait and there is an event
-- on the wire. Therefore the signal_wait outcome wires are the
-- true cause of satisfying the signal_wait.
WHERE
```

```
QUERY(w <* signal_wait_transaction |
     NOT EXISTS(w.outcome_cycle) OR
     (QUERY(ws <* wire_state |
           NOT(ws.parent_wire IN
               w.parent_wire_wait.sensitivity_signals) OR
           NOT(is_after(ws.sim_cycle,
                        w\wait_transaction.sim_cycle) AND
               is_before(ws.sim_cycle,w.outcome_cycle) AND
               ws.signal_event))
     = wire_state))
= signal_wait_transaction;
END_RULE;
```

The relationship **wait transaction - assign transaction**.

Satisfying a `wait transaction` can be seen as a cause of unblocking the transaction parent process and, therefore, of executing some of the wire assignments contained within the process. This relationship is modelled by the attributes `wait effect` and `effects`. Supplementary constraints are added to make sure that the cause-effect is constistent structurally.

```
ENTITY assign_transaction;
  wait_effect: OPTIONAL wait_transaction; ...
END_ENTITY;

ENTITY wait_transaction;
INVERSE
  effects: SET OF assign_transaction FOR wait_cause; ...
END_ENTITY;
```

The relationship **wait transaction - wait transaction**.

As in the case of the `wait transaction - assign transaction` relationship, satisfying a `wait transaction` can be seen as a cause of executing a wait statement contained within the same process. This relationship is modelled by the attributes `next wait` and `wait cause`.

```
ENTITY wait_transaction;
    next_wait: OPTIONAL wait_transaction;
INVERSE
    wait_cause: SET [0:1] OF wait_transaction FOR next_wait; ...
END_ENTITY;
```

Supplementary constraints are added to make sure that the cause-effect is constistent structurally.

However, this relationship is the base of an important global rule which models the 'blocking of a process'. The rule below states that in the simulation interval from a `wait transaction` to its effect (`next wait` if any), there must be no other `transaction` for the corresponding parent process.

```
RULE blocked_process FOR (wait_transaction,transaction);
-- No transaction for a process P can occur while a wait_transaction
-- for P is still in progress (not satisfied). P is blocked.
WHERE
QUERY(wt <* wait_transaction |
      NOT EXISTS(wt.outcome_cycle) OR
      (QUERY(t <* transaction |
            (wt\transaction.parent_real_process :<>:
             t.parent_real_process) OR
            NOT(is_after(t.sim_cycle,wt\transaction.sim_cycle) AND
                is_before(t.sim_cycle,wt.outcome_cycle)))
      = transaction))
= wait_transaction;
END_RULE;
```

An important aspect of modelling the satisfying process of a `wait transaction` relates to the `wait transaction outcome cycle`. The `outcome cycle` is the simulation cycle in which the `wait transaction` becomes satisfied. The modelling is different for the three types of waits.

The **outcome cycle** for a signal wait transaction is the simplest to model. It is the simulation cycle of the `wire state` which is the value of the attribute `outcome` of the `signal wait transaction`.

The **outcome cycle** for a delay wait transaction is the simulation cycle of the value of attribute `delay wait outcome` contained within a `delay wait transaction`. The value is an instance of an VHDL 'artificial' entity: `delay wait transaction outcome`. Satisfying a `delay wait transaction` relies solely on a time lapse. The end of this time lapse is signaled by a `delay wait transaction outcome` in the right simulation cycle (the first simulation cycle (from the simulation process) the `time value` of which equals the `time value` of the `delay wait transaction outcome` associated with the `delay wait transaction`).

The **outcome cycle** for a condition wait transaction is more difficult to explain. It relies on a computation process which involves wire states. As in the case of a `delay wait transaction` a VHDL 'artificial' entity is introduced: `condition wait value`. Each `real process` contains the full set of the condition wait values taken during the simulation process by all the predicates of the condition wait statements contained within the process. For each `condition wait` and for each `simulation cycle` there is such a value. In fact, the set of condition wait values for a given `condition wait` is similar to the set of wire states for a given wire.

The `outcome cycle` of a `condition wait transaction` is the simulation cycle of that `condition wait value` which (a) is associated with the wait transaction, (b) occurs after the wait transaction and c) is the 'first' such value `CV` with the attribute `CV.current value = VHDL true`. The rule below captures this rather difficult relationship.

```
RULE nearest_condition_wait_outcome FOR
     (condition_wait_transaction,condition_wait_value);
-- Between a condition_wait CW and its outcome_cycle, if this exists,
-- there is no VHDL_true condition_wait_value for CW based on events
-- on the condition predicate signals. Therefore, the value which
-- satisfies the condition_wait is the nearest such value.
WHERE
QUERY(w <* condition_wait_transaction |
      NOT EXISTS(w.outcome_cycle) OR
      (QUERY(v <* condition_wait_value |
            (v.parent_condition_wait :<>: w.parent_condition_wait) OR
            NOT(is_after(v.sim_cycle,w\wait_transaction.sim_cycle) AND
                is_before(v.sim_cycle,w.outcome_cycle) AND
                v.effective_value))
         = condition_wait_value))
= condition_wait_transaction;
END_RULE;
```

8 Modelling the VHDL Simulation Process

The behaviour schema outlines cause-effect relationships and other behavioural constraints on a one by one basis. In order to get a thorough image of the VHDL behavioural semantics all these separate aspects have to be integrated into a whole. This is achieved at the simulation level. Signal changes, statement execution, and therefore the generation of transactions, are scheduled by observing the discipline forced by the VHDL 'event-oriented' simulation strategy.

According to this strategy a signal cannot change its value in the same simulation cycle with the signal assignment which caused the change even if the change delay is zero. Similarly, a statement executed as the effect of a delay wait statement with a null delay cannot be in the same simulation cycle with the delay wait statement. Generally speaking any cause-effect activity must obey the law: the 'effect' simulation cycle must follow the 'cause' simulation cycle, even if the effect is instantaneous. This law makes the simulation process 'deterministic'.

The main entities of the simulation schema are wire and process 'histories', simulation cycles and the `simulation process`, as illustrated in figure 9.

The **simulation process** of an `architecture design` is the root of the behavioural model. It clusters the set of simulation cycles, the wire history and the process history, for all wires and all real processes within the simulated `architecture design`. In addition, it contains the set of `condition wait history` for all the condition waits in the simulated design. Moreover, there are two remarkable simulation cycles which mark the start and end of the simulation process.

```
ENTITY simulation_process;
  simulated_design: architecture_design;
  simulation_cycles: SET OF simulation_cycle;
```

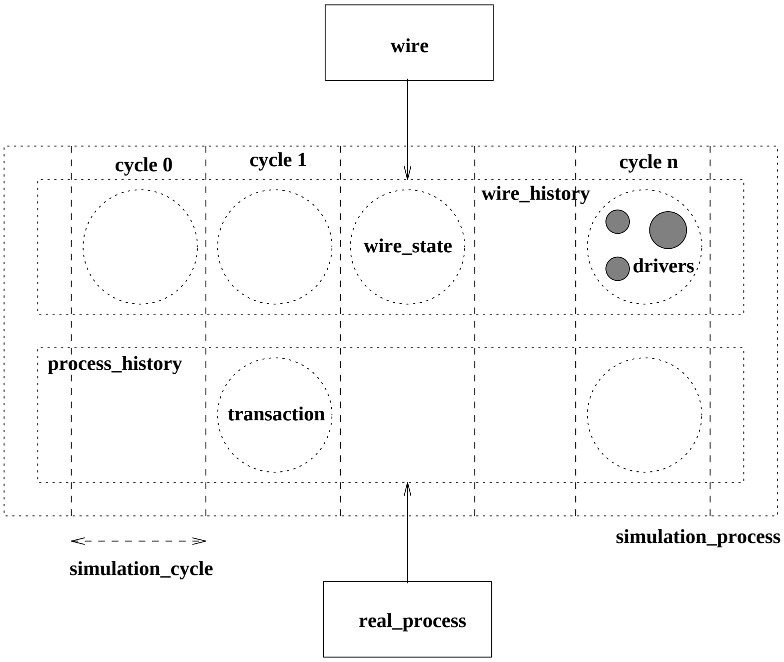

Figure 9: The simulation modelling level

```
  simulation_start_cycle: simulation_cycle;
  simulation_end_cycle: simulation_cycle;
  simulation_time_limit: OPTIONAL Time_value;

  wires_history: SET OF wire_history;
  processes_history: SET OF real_process_history;
  condition_waits_history: SET OF condition_wait_history;

DERIVE
  wires: SET OF wire:= simulated_design.wires;
  processes: SET OF real_process:= simulated_design.real_processes;
  simulation_start_time: Time_value:= simulation_start_cycle.event_time;
...
END_ENTITY;
```

The **wire history** groups all the states of a wire through the entire simulation process. There is a `wire state` in each simulation cycle of the simulation process.

The **process history** groups the transactions generated by executing the statements of a `real process`. Note that there can be simulation cycles with an empty set of transactions, when processes are blocked.

The **condition wait history** groups the values the predicate of a condition wait statement takes through the entire simulation process. There is a `condition wait value` for each condition wait statement and for each simulation cycle of the simulation process.

The **simulation cycle** clusters all the states of the wires and all the transactions of the processes of the simulated design at a given moment in time during the simulation process (see the constraints from the `simulation process` entity.) An important attribute of a simulation cycle is the earliest projected value `(evp)` of the assign drivers indirectly belonging to the cycle. It is used to determine the `time value` of the cycle. The `time value` of a simulation cycle is computed as follows:

- If the cycle contains delay wait outcomes then its time value is computed according to the delay transactions satisfied by the outcomes, i.e. the time of the cycle associated with such a transaction plus the delay time lapse.
- If there are projected values in the drivers of the simulation cycle and the cycle does not contain delay wait outcomes then the simulation cycle `time value` is the earliest projected value `(evp)`.
- If there are no projected values (i.e. all waveforms are empty) and no pending delay waits then the cycle must be the last cycle of the simulation process.

```
ENTITY simulation_cycle SUBTYPE OF(Event);
  wire_states: SET OF wire_state;
  transactions: SET OF transaction;
  condition_wait_values: SET OF condition_wait_value;
  delay_wait_outcomes: SET OF delay_wait_transaction_outcome;

DERIVE
  assign_drivers: SET OF assign_driver:=
  cycle_assign_drivers(wire_states);

  epv: projected_value:=
  earliest_projected_value(assign_drivers);

INVERSE
  parent_simulation: simulation_process FOR simulation_cycles;
 ...
END_ENTITY;
```

9 The VHDL Model

The following is an almost complete **core model** of VHDL at the levels of structure, configuration and elaboration, behaviour, simulation and time. It also incorporates an incomplete submodel of VHDL data types which, nevertheless, contains the information necessary at the other model levels.

The model does not tackle the computational aspects of VHDL. For example, the use of variables, computation oriented statements (variable assignments, conditionals, loops) and other algorithmic constructs are considered of marginal importance as far as the behavioural modelling of VHDL is concerned.

The schema boundaries within the model are commented. This is due to some problems the DEC EXPRESS checker had with multi-schema models at the time the model has been checked.

9.1 Auxiliary function schema

```
SCHEMA auxiliary_function_schema;

-- =====================================================
-- Auxiliary Express functions
-- =====================================================

 CONSTANT
        start_time: Time_value:= 0.0;
        -- The start of measurable time, not necessarily the
        -- simulation start time !

-- This constant belongs to time_schema. It is here only because
-- DEC EXPRESS is unable to work properly with multi-schema models.

        VHDL_true: VHDL_value:=
                   typed_object(boolean_type()) ||
                   VHDL_value() ||
                   enumeration_value(VHDL_identifier('TRUE'),integer_value(1));

-- This constant belongs to type_schema.
END_CONSTANT;

-- express type conversion, selection and test functions
FUNCTION is_type(ob: GENERIC; type_name: STRING): LOGICAL;
  RETURN(type_name IN TYPEOF(ob));
END_FUNCTION;

FUNCTION select_objects (type_name: STRING;
                         s: AGGREGATE OF GENERIC): AGGREGATE OF GENERIC;
  RETURN(QUERY (ob <* s | is_type(ob,type_name)));
END_FUNCTION;
```

```
FUNCTION conv_type(x: AGGREGATE OF GENERIC): AGGREGATE OF GENERIC;
  IF ? IN x THEN RETURN([]); END_IF;
  RETURN(x);
END_FUNCTION;
-- END_SCHEMA;
```

9.2 Generic block schema

```
-- SCHEMA generic_block_schema;
-- =======================================================
-- This is a model of block nesting and block object
-- visibility in general. This model will be specialized
-- to describe the VHDL architecture and block models.
-- =======================================================

ENTITY generic_object ABSTRACT SUPERTYPE OF(block_object);
END_ENTITY;

ENTITY generic_block ABSTRACT SUPERTYPE OF(VHDL_block);
  nested_blocks: SET OF generic_block;
  -- The blocks directly defined within SELF.

  local_objects: SET OF generic_object;
  -- The objects directly declared within SELF.

  parent_block: OPTIONAL generic_block;

DERIVE
  visible_objects:
  SET OF generic_object:= get_visible_objects(SELF);

WHERE
  tree_block_structure: no_cycles(SELF,nested_blocks);

  inverse_parent_block_for_nested_blocks:
  (QUERY(b <* nested_blocks | b.parent_block :=: SELF) = nested_blocks) AND
  block_in_parent_nested_blocks(SELF);
END_ENTITY;

FUNCTION block_in_parent_nested_blocks (b: generic_block): LOGICAL;
  IF NOT EXISTS(b.parent_block) THEN RETURN(TRUE); END_IF;
  RETURN(SELF IN b.parent_block.nested_blocks);
END_FUNCTION;

FUNCTION no_cycles (b: generic_block;
                    nested_blocks: SET OF generic_block): LOGICAL;
  IF nested_blocks = [] THEN RETURN(TRUE); END_IF;
  IF b IN nested_blocks THEN RETURN(FALSE); END_IF;
  RETURN(QUERY(nb <* nested_blocks |
                no_cycles(b,nb.nested_blocks)) = nested_blocks);
END_FUNCTION;
```

```
FUNCTION is_contained(c1,c2: generic_block): LOGICAL;
-- Tests is c1 is transitively contained within c2.

  IF NOT EXISTS(c1.parent_block) THEN RETURN (FALSE); END_IF;
  IF c1.parent_block :=: c2 THEN RETURN (TRUE); END_IF;
  RETURN (is_contained(c1.parent_block,c2));
END_FUNCTION;

FUNCTION get_visible_objects
         (b: generic_block): SET OF generic_object;
-- Computes the objects defined in the transitive parents of b

  IF NOT EXISTS(b.parent_block) THEN RETURN (b.local_objects); END_IF;
  RETURN (b.parent_block.local_objects +
          b.parent_block.visible_objects);
END_FUNCTION;

FUNCTION is_visible
         (ob: generic_object; b: generic_block): LOGICAL;
-- Tests if ob is declared within a block which contains b
-- directly or indirectly.

  RETURN (ob IN b.visible_objects);
END_FUNCTION;

-- END_SCHEMA;
```

9.3 Type schema

```
-- SCHEMA type_schema;
-- =======================================================
-- This is a model of the VHDL types and values. It is not
-- complete for an obvious reason: a type is characterized
-- not only by its set of values but, in addition, by a
-- set of operators and the semantics of these operators.
-- To model fully a data type means to model the semantics
-- of the operators. The model given here stops at the level
-- of the values corresponding to each type.

-- This schema is necessary since the model uses constraints
-- based on VHDL_true (for guarded concurrent assignments,
-- VHDL boolean_type, VHDL type compatibility etc. The VHDL
-- types are obviously isomorph but not identical to EXPRESS
-- types. The type BOOLEAN, for example, is a derived type
-- in VHDL (is an enumeration with the FALSE, TRUE values).
-- The type_schema explicits the VHDL -> EXPRESS type morphism.
-- =======================================================

-- The constant VHDL_true is normally here.

ENTITY VHDL_identifier;
```

```
  isomorph_to: STRING;
END_ENTITY;

ENTITY VHDL_value
       ABSTRACT SUPERTYPE OF (ONEOF(numerical_value,
                                    enumeration_value,
                                    record_value,
                                    array_of_values,
                                    array_value,
                                    access_value,
                                    file_of_values))
       SUBTYPE OF(typed_object);
END_ENTITY;

ENTITY numerical_value
       ABSTRACT SUPERTYPE OF (ONEOF(integer_value,floating_point_value))
       SUBTYPE OF(VHDL_value);
  isomorph_to: NUMBER;
END_ENTITY;

ENTITY VHDL_type
       ABSTRACT SUPERTYPE OF(ONEOF(user_type,basic_type));
  values: SET OF VHDL_value;
  -- The set of values characteristic to the VHDL_type.
  -- It will be specialised for each particular type.
END_ENTITY;

ENTITY basic_type
       ABSTRACT SUPERTYPE OF (ONEOF(scalar_type,composite_type,
                                    access_type,file_type))
       SUBTYPE OF(VHDL_type);
  id: STRING;
END_ENTITY;

ENTITY user_type
       ABSTRACT SUPERTYPE OF (ONEOF(composite_type,discrete_type,
                                    file_type,access_type))
       SUBTYPE OF(VHDL_type,block_object);
-- An user_type is a named_object, as all other block_objects
END_ENTITY;

ENTITY scalar_type
       ABSTRACT SUPERTYPE OF
       (ONEOF(discrete_type,integer_type,floating_point_type,physical_type))
       SUBTYPE OF(basic_type);
END_ENTITY;

ENTITY discrete_type
       ABSTRACT SUPERTYPE OF (ONEOF(integer_range_type,enumeration_type))
       SUBTYPE OF(scalar_type,user_type);
END_ENTITY;

ENTITY composite_type
       ABSTRACT SUPERTYPE OF(ONEOF(record_type,array_type))
```

```
       SUBTYPE OF(basic_type,user_type);
END_ENTITY;

ENTITY integer_type
       SUPERTYPE OF (integer_range_type)
       SUBTYPE OF(scalar_type);

  SELF\scalar_type.values: SET OF integer_value;
END_ENTITY;

RULE unique_integer_type FOR (integer_type);
WHERE SIZEOF(integer_type) = 1;
END_RULE;

ENTITY integer_value SUBTYPE OF(numerical_value);
  SELF\numerical_value.isomorph_to: INTEGER;

DERIVE
  SELF\numerical_value.object_type: integer:= parent_type;

INVERSE
  parent_type: integer_type FOR values;
END_ENTITY;

ENTITY integer_range_type
       SUBTYPE OF(discrete_type,integer_type);
  inf_limit,sup_limit: integer_value;

WHERE
  QUERY(v <* SELF\integer_type.values |
          {inf_limit.isomorph_to <= v.isomorph_to <= sup_limit.isomorph_to})
  = SELF\integer_type.values;
  -- The set of values of the integer_range_type
  -- has a specified size.
END_ENTITY;

ENTITY enumeration_type SUBTYPE OF(discrete_type,user_type);
  SELF\user_type.values: LIST [1 : ?] OF enumeration_value;

WHERE
  QUERY(ev <* values |
         SIZEOF(QUERY(ev1 <* values | ev.name = ev1.name)) = 1)
  = values;
  -- The names of the enumeration values are different.

  QUERY(ev <* values | ev.isomorph_to  =
                       index_of_value(ev,values))
  = values;
  -- An enumeration_value ev is isomorph with the (VHDL)
  -- integer_value 'equal' to the index of ev in the enumeration
  -- list of values. Note that index_of_value returns an
  -- (VHDL) integer_value.
END_ENTITY;
```

```
ENTITY enumeration_value SUBTYPE OF(VHDL_value);
  name: VHDL_identifier;
  isomorph_to: integer_value;

DERIVE
  SELF\VHDL_value.object_type: enumeration_type:= parent_type;

INVERSE
  parent_type: enumeration_type FOR values;
END_ENTITY;

FUNCTION index_of_value
         (e: enumeration_value;
          l: LIST OF enumeration_value): integer_value;

  LOCAL i: INTEGER; END_LOCAL;
  REPEAT i:= LOINDEX(l) TO HIINDEX(l) BY 1;
    IF l[i] = e THEN RETURN(integer_value(i)); END_IF;
  END_REPEAT;
END_FUNCTION;

ENTITY floating_point_type SUBTYPE OF(scalar_type);
  SELF\scalar_type.values: SET OF floating_point_value;
END_ENTITY;

ENTITY floating_point_value SUBTYPE OF(numerical_value);
  SELF\numerical_value.isomorph_to: REAL;

DERIVE
  SELF\VHDL_value.object_type: floating_point_type:= parent_type;

INVERSE
  parent_type: floating_point_type FOR values;
END_ENTITY;

RULE unique_floating_point_type FOR (floating_point_type);
WHERE SIZEOF(floating_point_type) = 1;
END_RULE;

ENTITY physical_type SUBTYPE OF(scalar_type,user_type);
  base_unit: physical_base_unit;
  secondary_units: SET OF physical_secondary_unit;
  SELF\user_type.values: SET OF physical_type_value;

WHERE
  QUERY(u1 <* secondary_units |
           (u1.unit_name <> base_unit.unit_name) AND
           (SIZEOF(QUERY(u2 <* secondary_units |
                            u1.unit_name = u2.unit_name)) = 1))
  = secondary_units;

  -- The base_unit and secondary units have different names.
END_ENTITY;
```

```
ENTITY physical_unit
       ABSTRACT SUPERTYPE OF
                (ONEOF(physical_base_unit,physical_secondary_unit));

  unit_name: VHDL_identifier;
  transform: numerical_value;
  -- It is considered that unit transformation is
  -- a function of the base_unit.
END_ENTITY;

ENTITY physical_base_unit SUBTYPE OF (physical_unit);
WHERE
  SELF\physical_unit.transform.isomorph_to = 1;
END_ENTITY;

ENTITY physical_secondary_unit SUBTYPE OF(physical_unit);
END_ENTITY;

ENTITY physical_type_value SUBTYPE OF(VHDL_value);
  unit: physical_unit;

DERIVE
  isomorph_to: NUMBER:= unit.transform.isomorph_to;
  SELF\VHDL_value.object_type: physical_type:= parent_type;

INVERSE
  parent_type: physical_type FOR values;
END_ENTITY;


ENTITY record_type SUBTYPE OF(composite_type);
  field_descriptors: SET [1 : ?] OF field_descriptor;
  SELF\composite_type.values: SET OF record_value;

WHERE
  QUERY(fd1 <* field_descriptors |
          SIZEOF(QUERY(fd2 <* field_descriptors |
                             fd1.field_id = fd2.field_id)) = 1)
  = field_descriptors;
  -- The names of fields are different.

  QUERY(v <* values |
        QUERY(fd <* field_descriptors |
                 SIZEOF(QUERY(fv <* v.field_value |
                                 fv.field = fd)) = 1)
        = field_descriptors)
  = values;
  -- A record_value is 'complete' as far as the record fields are
  -- concerned. Each field of the record_type is present in the value.
END_ENTITY;

ENTITY field_descriptor;
  field_id: VHDL_identifier;
  field_type: VHDL_type;
```

```
END_ENTITY;

ENTITY record_value SUBTYPE OF(VHDL_value);
  field_values: SET [1 : ?] OF field_value;

DERIVE
  SELF\VHDL_value.object_type: record_type:= parent_type;

INVERSE
  parent_type: record_type FOR values;
END_ENTITY;

ENTITY field_value;
  field: field_descriptor;
  current_value: VHDL_value;

WHERE
  is_same_type(current_value.object_type,field.field_type);
END_ENTITY;

ENTITY array_type
       ABSTRACT SUPERTYPE OF
                (ONEOF(unconstrained_array_type,constrained_array_type))
       SUBTYPE OF(composite_type);

  base_type: VHDL_type;
  index_constraints: LIST [1 : ?] OF index_constraint;
  SELF\composite_type.values: SET OF array_of_values;

  -- The values of an array_type are arrays of values of
  -- a given base_type.
END_ENTITY;

ENTITY array_of_values SUBTYPE OF(VHDL_value);
  array_elements: SET OF array_value;
  -- If the set is [] it means that no element is initialised.

DERIVE
  SELF\VHDL_value.object_type: array_type:= parent_type;

INVERSE
  parent_type: array_type FOR values;

WHERE
  QUERY(ev1 <* array_elements |
        SIZEOF(QUERY(ev2 <* array_elements |
                     ev1.indexes = ev2.indexes)) = 1)
  = array_elements;

  -- There is a single arrray_value for each position
  -- in the array.
END_ENTITY;

ENTITY array_value SUBTYPE OF(VHDL_value);
```

```
-- The value of an element of an array.

  indexes: LIST [1 : ?] OF integer_value;
  current_value: VHDL_value;

DERIVE
  SELF\VHDL_value.object_type: VHDL_type:=
  parent_subtype.parent_type.base_type;

INVERSE
  parent_subtype: array_of_values FOR array_elements;

WHERE
  SIZEOF(indexes) =
  SIZEOF(parent_subtype.parent_type.index_constraints);

  is_same_type(current_value.object_type,
               parent_subtype.parent_type.base_type);

  -- Also a constraint is necessary for saying that
  -- indexes are in range, according to the parent_array_type.
END_ENTITY;

ENTITY index_constraint;
  index_type: discrete_type;
  index_range: SET OF VHDL_value;

WHERE
  QUERY(ir <* index_range | is_same_type(index_type,ir.object_type))
  = index_range;

  (SIZEOF(index_range) = 2) OR (index_range = []);
END_ENTITY;

ENTITY unconstrained_array_type SUBTYPE OF(array_type);
WHERE
  QUERY(ic <* index_constraints | SIZEOF(ic.index_range) = 2) <> [];
  -- There are constraints on the range of array indexes
END_ENTITY;

ENTITY constrained_array_type SUBTYPE OF(array_type);
WHERE
  QUERY(ic <* index_constraints | SIZEOF(ic.index_range) > 0) = [];
  -- There are no constraints on the range of array indexes
END_ENTITY;

ENTITY access_type SUBTYPE OF(basic_type,user_type);
  base_type: VHDL_type;
  SELF\user_type.values: SET OF access_value;

WHERE
  NOT is_file_type(base_type);
  -- A dynamically allocated object cannot be a file.
END_ENTITY;
```

```
ENTITY access_value SUBTYPE OF(VHDL_value);
  current_value: VHDL_value;
DERIVE
  SELF\VHDL_value.object_type: access_type:= parent_type;

INVERSE
  parent_type: access_type FOR values;

WHERE
  is_same_type(current_value.object_type,parent_type.base_type);
END_ENTITY;

ENTITY file_type SUBTYPE OF(basic_type,user_type);
  base_type: VHDL_type;
  SELF\user_type.values: SET OF file_of_values;

  -- The values of a file_type are files_of_values of a given
  -- base_type.
WHERE
  -- Check if an access_type is member of the base_type.
  NOT access_type_in_VHDL_type(base_type);
END_ENTITY;

FUNCTION access_type_in_VHDL_type(t: GENERIC): LOGICAL;
-- Checks if t is an access_type or if it is build by
-- using access types.

  IF is_record_type(t)
  THEN RETURN(access_type_in_record_type(t));
  END_IF;

  IF is_array_type(t)
  THEN RETURN(access_type_in_array_type(t));
  END_IF;

  RETURN(is_access_type(t));
END_FUNCTION;

FUNCTION access_type_in_array_type(at: array_type): LOGICAL;
  RETURN(access_type_in_VHDL_type(at.base_type));
END_FUNCTION;

FUNCTION access_type_in_record_type(at: record_type): LOGICAL;
  LOCAL i: INTEGER; END_LOCAL;

  REPEAT i:= LOINDEX(at.field_descriptors) TO
             HIINDEX(at.field_descriptors) BY 1;
    IF access_type_in_VHDL_type(at.field_descriptors[i].field_type)
    THEN RETURN(TRUE);
    END_IF;
  END_REPEAT;

  RETURN(FALSE);
```

```
END_FUNCTION;

ENTITY file_of_values SUBTYPE OF(VHDL_value);
  file_elements: SET OF VHDL_value;

DERIVE
  SELF\VHDL_value.object_type: file_type:= parent_type;

INVERSE
  parent_type: file_type FOR values;

WHERE
  QUERY(e <* file_elements |
          is_same_type(e.object_type,parent_type.base_type))
  = file_elements;

  -- The type of file elements is the file base_type
END_ENTITY;

-- Derived types: natural, positive, boolean, bit, character etc.
-- All these are in fact constants of the integer_type and
-- enumeration_type.
-- ---------------------------------------------------------------

ENTITY natural_type SUBTYPE OF(integer_type);
WHERE
  QUERY(v <* SELF\integer_type.values | v.isomorph_to >= 0)
  = SELF\integer_type.values;
END_ENTITY;

ENTITY positive_type SUBTYPE OF(natural_type);
WHERE
  QUERY(v <* SELF\natural_type.values | v.isomorph_to <> 0)
  = SELF\natural_type.values;
END_ENTITY;

ENTITY boolean_type SUBTYPE OF(enumeration_type);
DERIVE
  SELF\enumeration_type.values: LIST OF enumeration_value:=
  [enumeration_value(VHDL_identifier('FALSE'),integer_value(0)),
   enumeration_value(VHDL_identifier('TRUE'),integer_value(1))];
END_ENTITY;

RULE unique_boolean_type FOR (boolean_type);
WHERE SIZEOF(boolean_type) = 1;
END_RULE;

ENTITY bit_type SUBTYPE OF(enumeration_type);
DERIVE
  SELF\enumeration_type.values: LIST OF enumeration_value:=
  [enumeration_value(VHDL_identifier('0'),integer_value(0)),
   enumeration_value(VHDL_identifier('1'),integer_value(1))];
END_ENTITY;
```

```
RULE unique_bit_type FOR (bit_type);
WHERE SIZEOF(bit_type) = 1;
END_RULE;

ENTITY character_type SUBTYPE OF(enumeration_type);
DERIVE
  SELF\enumeration_type.values: LIST OF enumeration_value:=
  [enumeration_value(VHDL_identifier('a'),integer_value(65)),
   enumeration_value(VHDL_identifier('b'),integer_value(66))]; -- etc.
END_ENTITY;

RULE unique_character_type FOR (character_type);
WHERE SIZEOF(bit_type) = 1;
END_RULE;

-- Etc. other derived predefined types can be described.
-- However, they are not important for modelling the structural
-- and behavioural aspects of VHDL.

FUNCTION is_same_type(t1,t2: VHDL_type): LOGICAL;
  RETURN(t1 = t2);
END_FUNCTION;

FUNCTION is_boolean_type(t: VHDL_type): LOGICAL;
  RETURN(is_type(t,'TYPE_SCHEMA.BOOLEAN_TYPE'));
END_FUNCTION;

FUNCTION is_file_type(t: VHDL_type): LOGICAL;
  RETURN(is_type(t,'TYPE_SCHEMA.FILE_TYPE'));
END_FUNCTION;

FUNCTION is_access_type(t: VHDL_type): LOGICAL;
  RETURN(is_type(t,'TYPE_SCHEMA.ACCESS_TYPE'));
END_FUNCTION;

FUNCTION is_array_type(t: VHDL_type): LOGICAL;
  RETURN(is_type(t,'TYPE_SCHEMA.ARRAY_TYPE'));
END_FUNCTION;

FUNCTION is_record_type(t: VHDL_type): LOGICAL;
  RETURN(is_type(t,'TYPE_SCHEMA.RECORD_TYPE'));
END_FUNCTION;

FUNCTION is_predefined(t: VHDL_type): LOGICAL;
  RETURN(NOT is_type(t,'TYPE_SCHEMA.USER_TYPE'));
END_FUNCTION;

FUNCTION is_type_compatible(t1,t2: VHDL_type): LOGICAL;
-- The result is over simplified at this stage.
-- This function should implement type compatibility rules
-- from VHDL.
  RETURN(t1 = t2);
END_FUNCTION;
```

```
-- END_SCHEMA;
```

9.4 Block object schema

```
-- SCHEMA block_object_schema;

-- ======================================================
-- This schema models the atomic objects which can be
-- contained within an architecture/block specialization
-- of a generic_block. Signal, port and generic const.
-- are specializations of generic_object.

-- Atomic objects are 'named' and/or 'typed'.
-- By capturing the name of an atomic object in the model
-- makes it easier to model block containment of signals,
-- ports and generic constants with identical properties
-- (and different names).
-- ======================================================

-- typed and named objects
-- -----------------------

ENTITY typed_object
       ABSTRACT SUPERTYPE OF
       (ONEOF(atomic_object, VHDL_value, VHDL_function,
              signal_or_port_source));

  object_type: VHDL_type;
END_ENTITY;

ENTITY named_object
       ABSTRACT SUPERTYPE OF
       (ONEOF(block_object,library,library_unit));
  id: VHDL_identifier;
END_ENTITY;

ENTITY block_object
       ABSTRACT SUPERTYPE OF (ONEOF(real_block_object,
                             atomic_object, user_type,
                             component, VHDL_function))
       SUBTYPE OF (generic_object,named_object);
END_ENTITY;

FUNCTION test_for_defined_types
         (user_types: SET OF VHDL_type;
          typed_objects: SET OF typed_object): LOGICAL;

-- Tests if the types of typed_objects are in user_types
RETURN(QUERY(ob <* typed_objects |
                is_predefined(ob.object_type) OR
                (ob.object_type IN user_types))
       = typed_objects);
```

```
END_FUNCTION;

--  resolution_function, conversion_function, component
-- ---------------------------------------------------

ENTITY VHDL_function
       ABSTRACT SUPERTYPE OF (ONEOF(function_instance,
                             resolution_function,
                             conversion_function))
       SUBTYPE OF(block_object,typed_object);
END_ENTITY;

ENTITY resolution_function SUBTYPE OF(VHDL_function);
  no_of_parameters: INTEGER;
END_ENTITY;

ENTITY conversion_function SUBTYPE OF(VHDL_function);
  converted_type: VHDL_type;
END_ENTITY;

ENTITY component SUBTYPE OF(block_object);
  ports: SET OF port;
  generics: SET OF generic_constant;
END_ENTITY;

-- atomic_object: signals, ports and generic_constants
-- ---------------------------------------------------

ENTITY atomic_object
       ABSTRACT SUPERTYPE OF (ONEOF(real_atomic_object,
                             signal_or_port,generic_constant))
       SUBTYPE OF(block_object,typed_object);
  default_value: VHDL_value;

WHERE
  valid_value:
  is_type_compatible(default_value.object_type,object_type);
END_ENTITY;

ENTITY signal_or_port
       ABSTRACT SUPERTYPE OF(ONEOF(wire,signal,port))
       SUBTYPE OF(atomic_object);

  sources: SET OF signal_or_port_source;

  -- The signal/port can have multiple sources:
  --  a) being connected to other signals and ports
  --  b) being controlled by several processes via assignments

  resolved_by: OPTIONAL resolution_function;
  -- The resolution function of the signal must be in the
  -- same architecture as the parent block of the signal
  -- (see architecture_block)
```

```
DERIVE
  is_resolved: LOGICAL:= EXISTS(resolved_by);

WHERE
  resolution_condition:
  ((SIZEOF(sources) > 1) AND EXISTS(resolved_by)) OR
  ((SIZEOF(sources) = 0) AND NOT EXISTS(resolved_by)) OR
  (SIZEOF(sources) = 1);

  -- The resolution function must be present if connections > 1
  -- and absent if the signal/port is unconnected. However it
  -- can exist or not if SIZEOF(connections) = 1.

  buffer_ports_cannot_have_multiple_sources:
  NOT is_type(SELF,'STRUCTURE_SCHEMA.BUFFER_PORT') OR
  (SIZEOF(sources) <= 1);

  valid_resolution_function: test_resolution_function(SELF);

  -- If SELF is resolved the no. of the sources must be
  -- compatible with the no. of resolution function parameters
  -- and the resolution function type must be compatible with
  -- the type of SELF.

  inverse_target_for_sources:
  QUERY(s <* sources | s.target :=: SELF) = sources;
END_ENTITY;

ENTITY signal_or_port_source
       ABSTRACT SUPERTYPE OF(ONEOF(wire_source,
                            port_connection,signal_assignment))
       SUBTYPE OF(typed_object);

  -- The attribute object_type is inherited from typed_object.
  -- It is the type of the value the source produces.
  target: signal_or_port;

WHERE
  SELF IN target.sources;
  is_type_compatible(object_type,target.object_type);

  is_writable(target);
  -- Only a writable signal can be driven.
END_ENTITY;

FUNCTION test_resolution_function
         (s: signal_or_port): LOGICAL;

  IF s.is_resolved
  THEN RETURN ((s.resolved_by.no_of_parameters =
              SIZEOF(s.sources)) AND
             is_type_compatible(s.object_type,
                                s.resolved_by.object_type));
  ELSE RETURN(TRUE);
```

```
  END_IF;
END_FUNCTION;

ENTITY signal
       ABSTRACT SUPERTYPE OF(ONEOF(simple_signal,bus,register))
       SUBTYPE OF(signal_or_port,readable,writable);
END_ENTITY;

ENTITY simple_signal SUBTYPE OF(signal);
END_ENTITY;

ENTITY bus SUBTYPE OF(signal);
END_ENTITY;

ENTITY register SUBTYPE OF(signal);
END_ENTITY;

ENTITY port
       ABSTRACT SUPERTYPE OF (ONEOF(in_port,out_port,
                                    inout_port,buffer_port))
       SUBTYPE OF(signal_or_port);
END_ENTITY;

ENTITY in_port SUBTYPE OF(port,readable);
END_ENTITY;

ENTITY out_port SUBTYPE OF(port,writable);
END_ENTITY;

ENTITY inout_port SUBTYPE OF(port,readable,writable);
END_ENTITY;

ENTITY buffer_port SUBTYPE OF(port,readable,writable);
END_ENTITY;

ENTITY readable;
END_ENTITY;

ENTITY writable;
END_ENTITY;

FUNCTION is_readable(x: signal_or_port): LOGICAL;
  RETURN(is_type(x,'BLOCK_OBJECT_SCHEMA.READABLE'));
END_FUNCTION;

FUNCTION is_writable(x: signal_or_port): LOGICAL;
  RETURN(is_type(x,'BLOCK_OBJECT_SCHEMA.WRITABLE'));
END_FUNCTION;

FUNCTION is_signal(x: signal_or_port): LOGICAL;
  RETURN(is_type(x,'BLOCK_OBJECT_SCHEMA.SIGNAL'));
END_FUNCTION;

FUNCTION is_bus(x: signal_or_port): LOGICAL;
```

```
  RETURN(is_type(x,'BLOCK_OBJECT_SCHEMA.BUS'));
END_FUNCTION;

FUNCTION is_register(x: signal_or_port): LOGICAL;
  RETURN(is_type(x,'BLOCK_OBJECT_SCHEMA.REGISTER'));
END_FUNCTION;

FUNCTION is_port(x: signal_or_port): LOGICAL;
  RETURN(is_type(x,'BLOCK_OBJECT_SCHEMA.PORT'));
END_FUNCTION;

FUNCTION is_guarded_signal(s: signal_or_port): LOGICAL;
  RETURN(is_bus(s) OR is_register(s));
END_FUNCTION;

ENTITY generic_constant SUBTYPE OF(atomic_object);
  const_value: VHDL_value;

WHERE
  valid_constant_value:
  is_type_compatible(object_type,const_value.object_type);
END_ENTITY;

-- END_SCHEMA;
```

9.5 Structure schema

```
-- SCHEMA structure_schema;
-- =======================================================
-- This is a model of VHDL architecture and block, package,
-- VHDL entity and  concurrent statement. The model of VHDL
-- architecture/block is obtained by specializing the
-- generic_block in order to contain (VHDL) block_objects
-- as specified by the block_object_schema.

-- Note that all the entities defined at this level of
-- modelling behave as templates. For example, a signal
-- can well be in the set of local_objects of several
-- architectures. That is possible since, at this level,
-- signals and ports do not 'contain' current, time dependent
-- values.

-- The focus of modelling at this level is structure:
-- containment, visibility, signal-port conections and
-- component instantiation. At the level of design_unit_schema,
-- signals and ports will be specialized to wires which
-- contain states and so on. Still a looong way to go.
-- =======================================================

-- package_declaration, package_body, entity_declaration
--_____________________________________________________
```

```
ENTITY declarative_unit
       ABSTRACT SUPERTYPE OF
       (ONEOF(package_declaration,entity_declaration,VHDL_block));

  user_types: SET OF VHDL_type;
  signals: SET OF signal;
  functions: SET OF VHDL_function;
  components: SET OF component;
  used_packages: SET OF package_declaration;

WHERE
  -- The user_types, signals, resolution_functions,
  -- (type) conversion_functions and components
  -- usable within SELF are either locally
  -- declared or inherited from the used_packages.

  inherited_types_from_packages:
  QUERY(pk <* used_packages | pk.user_types <= user_types)
  = used_packages;

  inherited_functions_from_packages:
  QUERY(pk <* used_packages | pk.functions <= functions)
  = used_packages;

  inherited_signals_from_packages:
  QUERY(pk <* used_packages | pk.signals <= signals)
  = used_packages;

  inherited_components_from_packages:
  QUERY(pk <* used_packages |
           pk.components <= components) = used_packages;

  -- The user_types of the package signals, component
  -- ports and generics, and resolution functions
  -- must be defined locally or inherited from the
  -- packages used within SELF.

  check_signal_types:
  test_for_defined_types(user_types,signals);

  check_function_types:
  QUERY(f <* functions | test_for_defined_types(user_types,[f]))
  = functions;

  check_component_types:
  QUERY(c <* components |
          test_for_defined_types(user_types,c.ports) AND
          test_for_defined_types(user_types,c.generics))
  = components;

  -- The types of signals, functions and component ports and
  -- generics must be predefined or, in the case of user types,
  -- they have to be visible from the point of their use.
END_ENTITY;
```

```
ENTITY package_declaration
       SUBTYPE OF(declarative_unit,library_unit);
WHERE
  no_cyclic_use: no_cyclic_package_use(SELF,used_packages);
END_ENTITY;

ENTITY package_body SUBTYPE OF(library_unit);
  package_decl: package_declaration;
END_ENTITY;

FUNCTION no_cyclic_package_use
         (p: package_declaration;
          used_packages: SET OF package_declaration):
LOGICAL;
  IF used_packages = [] THEN RETURN(TRUE); END_IF;
  IF p IN used_packages THEN RETURN(FALSE); END_IF;
  RETURN(QUERY(up <* used_packages |
                  no_cyclic_package_use(p,up.used_packages))
         = used_packages);
END_FUNCTION;

ENTITY entity_declaration
       SUBTYPE OF(declarative_unit,library_unit);

  ports: SET OF port;
  generics: SET OF generic_constant;

WHERE
  no_component_declarations:
  SELF\declarative_unit.components = [];
END_ENTITY;

--  architecture_body, block
-- -------------------------

ENTITY VHDL_block
       SUPERTYPE OF(ONEOF(VHDL_block_config,block,architecture_body))
       SUBTYPE OF(generic_block,declarative_unit);

  -- specialization of generic_block attributes
  SELF\generic_block.nested_blocks: SET OF block;
  SELF\generic_block.parent_block: OPTIONAL VHDL_block;
  SELF\generic_block.local_objects: SET OF block_object;

  -- additional attributes
  processes: BAG OF process;
  component_instances: BAG OF component_instance;

  -- Note BAG ! For example there can be several identical
  -- concurrent assignments in a block or architecture.

  generics: SET OF generic_constant;
  ports: SET OF port;
```

```
  guard: OPTIONAL simple_signal;
  parent_architecture: architecture_body;

  signals_and_ports: SET OF signal_or_port;
  concurrent_statements: BAG OF concurrent_statement;

DERIVE
  has_guard: LOGICAL:= EXISTS(guard);

WHERE
  signals_and_ports = conv_type(signals) +
                      conv_type(ports) +
                      conv_type([guard]);

  concurrent_statements =
                      conv_type(processes) +
                      conv_type(component_instances) +
                      conv_type(nested_blocks);

  SELF\generic_block.local_objects =
                      conv_type(user_types) +
                      conv_type(components) +
                      conv_type(functions) +
                      conv_type(generics) +
                      conv_type(signals_and_ports);

  -- The local_objects must be exactly those from the
  -- declarative_unit + ports + generics. This forces
  -- the local_objects of the block which are part
  -- of the declarative_unit to obey the constraints
  -- from the declarative_unit.

  concurrent_statements_is_a_bag_of_values:
  QUERY(x <* concurrent_statements |
          SIZEOF(QUERY(y <* concurrent_statements | x :=: y)) = 1)
  = concurrent_statements;

  -- It is not clear from the EXPRESS textbook whether a BAG is
  -- a BAG of values and a SET of instances or if it is a BAG
  -- of both values and instances. The above constraint makes it
  -- clear that the bag of concurrent statements is only a bag
  -- of values i.e. all the elements in the BAG are different
  -- (:<>:) although their values can be equal (=). This property
  -- makes it easier to specify correctly the constraints on bags
  -- of concurrent statements (for example, that all the components
  -- of a block must be configured in a block configuration item)

  resolution_functions_are_visible:
  QUERY(sp <* signals_and_ports |
           NOT sp.is_resolved OR is_visible(sp.resolved_by,SELF))
  = signals_and_ports;

  -- For all signal_or_ports the resolution function,
```

```
  -- if present, must be visible from SELF.

  check_port_types: test_for_defined_types(user_types,ports);

  -- The types of ports must be predefined or, in the case
  -- of user types, they have to be visible from the point
  -- of their use.

  boolean_guard_signal: check_guard_signal(SELF);
  -- The guard_signal must be boolean

   ('LIBRARY_SCHEMA.VHDL_BLOCK_CONFIG' IN TYPEOF( SELF) ) OR
   (SIZEOF(QUERY(c <* component_instances |
                'LIBRARY_SCHEMA.COMPONENT_CONFIG' IN TYPEOF(c))) = 0);

  -- Configurated component instances can appear only within
  -- configured architectures and blocks.

  inverse_parent_block_for_concurrent_statements:
  SIZEOF(QUERY(c <* concurrent_statements | c.parent_block :=: SELF)) =
  SIZEOF(concurrent_statements);
END_ENTITY;

FUNCTION check_guard_signal(b: VHDL_block): LOGICAL;
  IF b.has_guard
  THEN RETURN(is_boolean_type(b.guard.object_type));
  ELSE RETURN(TRUE);
  END_IF;
END_FUNCTION;

ENTITY architecture_body
       SUPERTYPE OF(architecture_config)
       SUBTYPE OF(VHDL_block,library_unit);

  -- An architecture is considered as a VHDL_block
  -- containing concurrent statements. For any object
  -- ob of these:
  --    a) ob.parent_block :=: SELF and, in addition,
  --    b) ob.parent_block.parent_architecture :=: SELF.

  -- The ports, signals and generic constants usable
  -- within SELF can be inherited from the associated
  -- entity_declaration.

  architecture_entity: entity_declaration;

DERIVE
  SELF\VHDL_block.ports: SET OF port:=
  architecture_entity.ports;

  SELF\VHDL_block.generics: SET OF generic_constant:=
  architecture_entity.generics;

WHERE
```

```
  parent_architecture :=: SELF;

  NOT EXISTS(SELF\VHDL_block.parent_block);

  inherited_signals_from_entity:
  architecture_entity.signals <= signals;
END_ENTITY;


ENTITY block
       SUPERTYPE OF(block_config)
       SUBTYPE OF(VHDL_block,concurrent_statement);
  port_map: SET OF port_connection;

DERIVE
  SELF\VHDL_block.parent_block: VHDL_block:=
  SELF\concurrent_statement.parent_block;

WHERE
  parent_architecture :=: SELF\concurrent_statement.parent_architecture;

  is_contained(SELF, parent_architecture);

  block_ports_are_mapped:
  QUERY(connection <* port_map |
                 connection.locall IN ports) = port_map;
  -- The locals in the port map must be the ports of the SELF block

  block_ports_are_connected_to_visible_signals_or_ports:
  QUERY(connection <* port_map |
                  is_visible(connection.actual,SELF))
  = port_map;

  -- The actuals in the port map must be signals or ports
  -- visible from SELF
END_ENTITY;

--  concurrent_statements: component_instances and processes
-- ---------------------------------------------------------

ENTITY concurrent_statement ABSTRACT SUPERTYPE OF
       (ONEOF(real_concurrent_statement,
              component_instance,process,block));
  parent_block: VHDL_block;

DERIVE
  parent_architecture:
  architecture_body:= parent_block.parent_architecture;

WHERE SELF IN parent_block.concurrent_statements;
END_ENTITY;

-- component instances
-- -------------------
```

```
ENTITY component_instance SUPERTYPE OF(component_config)
                          SUBTYPE OF(concurrent_statement);
  instantiated_component: component;
  port_map: SET OF port_connection;
  generic_map: SET OF generic_aspect;

WHERE
  is_visible(instantiated_component,
             SELF\concurrent_statement.parent_block);

  component_ports_are_mapped:
  QUERY(connection <* port_map |
        connection.locall IN instantiated_component.ports)
  = port_map;
  -- The ports in the port map must be the ports of
  -- the instantiated component.

  component_ports_are_connected_to_visible_signals:
  QUERY(connection <* port_map |
                    is_visible(connection.actual,parent_block))
  = port_map;

  -- The actuals in the port map must be signals or ports
  -- visible from the component_instance's parent_block.

  entity_ports_as_actuals:
  QUERY(connection <* port_map |
        is_signal(connection.actual) OR
        (connection.actual IN conv_type(parent_architecture.ports)))
  = port_map;

  -- The actuals in the port map must be signals or ports of
  -- the entity associated with the parent_architecture.

  valid_port_connections:
  (SIZEOF(port_map) = SIZEOF(instantiated_component.ports)) AND
  (QUERY(p <* instantiated_component.ports |
           SIZEOF(QUERY(pc <* port_map | p = pc.locall)) = 1)
   = instantiated_component.ports);

  -- There is a one to one correspondence between the ports
  -- of the instantiated component and the port_map connections.

  some_generic_const_are_redefined:
  QUERY(ga <* generic_map |
           ga.generic_const IN instantiated_component.generics)
  = generic_map;

  -- Some generic constants defined within the
  -- instantiated component can be redefined as
  -- far as their values are concerned.

  conversion_functions_are_visible:
```

```
  QUERY(con <* port_map |
           NOT con.is_converted OR
           is_visible(con.converted_by,parent_block))
  = port_map;

  -- For all connections of the port_map which are
  -- subject to a type conversion, the conversion function
  -- must be visible from the parent_block of the component
  -- instance.
END_ENTITY;

ENTITY generic_aspect;
  generic_const: generic_constant;
  assoc_value: VHDL_value;

WHERE
  valid_value_type:
  is_type_compatible(generic_const.object_type,
                     assoc_value.object_type);
END_ENTITY;

ENTITY port_connection SUBTYPE OF (signal_or_port_source);
  locall: port;
  actual: signal_or_port;
  -- The local is always a port of a block or of an
  -- instantiated component. The actual can be a both
  -- signal or a port.

  source: signal_or_port;
  -- The source and target model the connection
  -- (locall => actual) from the point of view (driver
  -- signal -> driven signal). Note that
  -- [source,target] * [locall,actual] = [locall,actual]

  --  target: is inherited from signal_or_port_source
  -- This is the driven signal by force of connection

  converted_by: OPTIONAL conversion_function;

DERIVE
  is_converted: LOGICAL:= EXISTS(conversion_function);

WHERE
  distinct_source_target: source :<>: target;
  -- Not quite sure about that. Can an in_out port be
  -- connected to itself ?
  -- If the answer is yes this constraint vanishes.

  local_is_one_of_source_or_target:
  (locall :=: target) OR (locall :=: source);

  actual_is_one_of_source_or_target:
  (actual :=: target) OR (actual :=: source);
```

```
  source_can_be_read: is_readable(source);

  type_of_source: check_conversion_type(SELF);

  -- The connection acts as one of the signal sources for the
  -- connection target signal. The type of this source is:
  --  a) the type of the conversion function (if the connection
  --     source is type converted,
  --  b) the type of the source signal, if there is no
  --     type conversion.
END_ENTITY;

FUNCTION check_conversion_type (con: port_connection): LOGICAL;

-- Checks if the type of a connection is either
-- the type of the connection source or the type
-- of the conversion function which converts the
-- type of the connection source.

  IF con.is_converted
  THEN RETURN(is_type_compatible(con.converted_by.object_type,
                                 con.object_type) AND
              is_type_compatible(con.converted_by.converted_type,
                                 con.source.object_type));
  ELSE RETURN(is_type_compatible(con.source.object_type,
                                 con.object_type));
  END_IF;
END_FUNCTION;

-- processes and concurrent assignments
-- ------------------------------------

ENTITY process
       ABSTRACT SUPERTYPE OF (ONEOF(process_statement,
                                    concurrent_assignment))
       SUBTYPE OF(concurrent_statement);

  -- Models a process_statement or
  -- a concurrent_assignment_statement.

  consulted_signals: SET OF signal_or_port;
  -- Signals used in local: control, wait and signal assignments
  -- from within the process.

  assignments: SET OF signal_assignment;
  delay_waits: SET OF delay_wait_statement;
  signal_waits: SET OF signal_wait_statement;
  condition_waits: SET OF condition_wait_statement;

  is_guarded: LOGICAL;
  -- Shows if the signal_assignments are guarded. (This can happen
  -- only for a concurrent_assignment).

DERIVE
```

```
  guard_signal: signal:= SELF\concurrent_statement.parent_block.guard;
  -- The signal which guards the assignments from within the process.

  sensitivity_signals: SET OF signal_or_port:=
  signals_from_signal_waits(signal_waits);

  controlled_signals:
  SET OF signal_or_port:= assignment_target_signals(assignments);

  assign_signals: SET OF signal_or_port:=
  assignment_signals(assignments);
  -- Signals used in the right part of the process assignments.

  condition_signals: SET OF signal_or_port:=
  signals_from_condition_waits(condition_waits);

  signals: SET OF signal_or_port:= controlled_signals +
                                   sensitivity_signals +
                                   consulted_signals;
  -- The set of all signals used within the process.

WHERE
  visible_signals:
  QUERY(s <* signals | is_visible(s,parent_block)) = signals;
  -- Signals must be visible to the process parent_block.

  sensitivity_signals_must_be_readable:
  QUERY(s <* sensitivity_signals | is_readable(s))
  = sensitivity_signals;

  consulted_signals_must_be_readable:
  QUERY(s <* consulted_signals | is_readable(s))
  = consulted_signals;

  -- Controlled signals are writable by force of
  -- the constraint within the signal_or_port_source
  -- which tells that the target of the source must
  -- be writable.

  consulted_signals_membership:
  (assign_signals + sensitivity_signals + condition_signals)
  <= consulted_signals;

  -- Note that the consulted signals can contain additional
  -- signals used in control statements within the process.

  unique_targets:
  SIZEOF(assignments) = SIZEOF(controlled_signals);

  -- Only one synthetic assignment for each controlled signal.
  -- This assignment represents all textual signal assignments
  -- contained within the process for that signal/port.

  guarded_assignments_must_be_in_a_block_with_guard:
```

```
  (NOT is_guarded) OR SELF\concurrent_statement.parent_block.has_guard;
END_ENTITY;


ENTITY signal_assignment SUBTYPE OF(signal_or_port_source);
-- A signal assignment is like a virtual connection.
-- It drives the target signal by force of a computed value
-- the type of which is assigned_value_type. Whether
-- the target is resolved or not the assigned_value_type
-- must be compatible with the type of the target_signal.

--  target and assigned_value are inherited from
--  signal_or_port_source

 assignment_signals: SET OF signal_or_port;
 -- target <= expression of assignment_signals
END_ENTITY;

ENTITY delay_wait_statement;
  delay: Time_value;

WHERE
  delay >= 0;
END_ENTITY;

ENTITY signal_wait_statement;
  sensitivity_signals: SET [1 : ?] OF signal_or_port;
END_ENTITY;

ENTITY condition_wait_statement;
  condition_signals: SET [1 : ?] OF signal_or_port;
END_ENTITY;

ENTITY concurrent_assignment SUBTYPE OF(process);

-- A concurrent_assignment is a process with no
-- delay_waits, a single synthetic signal_assignment
-- (a single target signal) and at most one signal_wait
-- based on the guard signal.

WHERE
  no_delay_waits: delay_waits = [];
  no_condition_waits: condition_waits = [];

  SIZEOF(signal_waits) <= 1;
  SIZEOF(assignments) = 1;

  sensitivity_signals = [SELF\process.guard_signal\signal_or_port];
END_ENTITY;

ENTITY process_statement SUBTYPE OF(process);

-- A process_statement is a process with unguarded
-- assignments (signal sequential assignments). There
```

```
-- can be several synthetic assignments (several target
-- signals) within the process_statement.

WHERE
  NOT(is_guarded);
END_ENTITY;

FUNCTION assignment_target_signals
         (assign: SET OF signal_assignment): SET OF signal_or_port;

  LOCAL i: INTEGER:= LOINDEX(assign);
        s: SET OF signal_or_port:= [];
  END_LOCAL;

  REPEAT i:= LOINDEX(assign) TO HIINDEX(assign) BY 1;
         s := s + [assign[i].target];
  END_REPEAT;
  RETURN(s);
END_FUNCTION;

FUNCTION assignment_signals
         (assign: SET OF signal_assignment): SET OF signal_or_port;

  LOCAL i: INTEGER:= LOINDEX(assign);
        s: SET OF signal_or_port:= [];
  END_LOCAL;

  REPEAT i:= LOINDEX(assign) TO HIINDEX(assign) BY 1;
         s := s + assign[i].assignment_signals;
  END_REPEAT;
  RETURN(s);
END_FUNCTION;

FUNCTION signals_from_signal_waits
         (w: SET OF signal_wait_statement):
         SET OF signal_or_port;

  LOCAL i: INTEGER;
        s: SET OF signal_or_port:= [];
  END_LOCAL;

  REPEAT i:= LOINDEX(w) TO HIINDEX(w) BY 1;
      s:= s + w[i].sensitivity_signals;
  END_REPEAT;
  RETURN(s);
END_FUNCTION;

FUNCTION signals_from_condition_waits
         (w: SET OF condition_wait_statement):
         SET OF signal_or_port;

  LOCAL i: INTEGER;
        s: SET OF signal_or_port:= [];
  END_LOCAL;
```

```
  REPEAT i:= LOINDEX(w) TO HIINDEX(w) BY 1;
      s:= s + w[i].condition_signals;
  END_REPEAT;
  RETURN(s);
END_FUNCTION;

-- END_SCHEMA;
```

9.6 Library schema

```
-- SCHEMA library_schema;
-- ======================================================
-- This is a model of the library unit. Apart from package
-- body and declaration, architecture body and entity decl.,
-- modelled by the structure_schema, the configuration
-- statements are described.

-- The configuration of a component (component_config) is
-- seen as the specialization of a component_instance, from
-- the structure level.

-- The configuration of a block is described as the
-- specialization of the VHDL_block.

-- In this way the structural constraints are inherited
-- from the structure_schema and the model is simplified.
-- ======================================================

ENTITY library_unit
       ABSTRACT SUPERTYPE OF
       (ONEOF(entity_declaration,architecture_body,
              package_declaration,package_body,
              configuration_declaration))
       SUBTYPE OF(named_object);
INVERSE
  owner_library: library FOR library_units;
END_ENTITY;

ENTITY library SUBTYPE OF(named_object);
  library_units: SET OF library_unit;

DERIVE
  enitity_declarations: SET OF entity_declaration:=
    select_objects('STRUCTURE_SCHEMA.ENTITY_DECLARATION',
                   library_units);

  architecture_bodies: SET OF architecture_body:=
    select_objects('STRUCTURE_SCHEMA.ARCHITECTURE_BODY',
                   library_units);

  package_declarations: SET OF package_declaration:=
```

```
    select_objects('STRUCTURE_SCHEMA.PACKAGE_DECLARATION',
                   library_units);

  package_bodies: SET OF package_body:=
    select_objects('STRUCTURE_SCHEMA.PACKAGE_BODY',
                   library_units);

  configuration_declarations: SET OF configuration_declaration:=
    select_objects('LIBRARY_SCHEMA.CONFIGURATION_DECLARATION',
                   library_units);
WHERE
  unique_names_within_library:
  QUERY(unit1 <* library_units |
        (SIZEOF(QUERY(unit2 <* library_units |
                   (unit1.id = unit2.id) AND
                   (TYPEOF(unit1) = TYPEOF(unit2)))) = 1))
  = library_units;
  -- Units of the same type have different names.
END_ENTITY;

ENTITY configuration_declaration SUBTYPE OF(library_unit);
  configuration_items: SET OF configuration_item;
END_ENTITY;

ENTITY configuration_item
       ABSTRACT SUPERTYPE OF(ONEOF(VHDL_block_config,
                                   component_config));

-- It is important to remember that at this level of modelling
-- signals, ports, entities and architectures are templates.
-- block_config and component_config are also templates which
-- show how to 'expand' the component instantiations from a
-- given architecture or block in terms of other architectures
-- and entity declarations.

-- At the level of design_schema modelling, a block_config entity,
-- which recursively references component_configs, will be used
-- to build the model of a simulatable design. This model
-- works with 'real' signals and ports which are instances of
-- the signal/port templates from the structure model level.
END_ENTITY;

ENTITY component_config
       SUBTYPE OF(configuration_item,component_instance);

-- This is the model of the configuration of a component_instance.

-- Let instantiated_component be the component instantiated
-- according to the component_instance which is configurated.
-- Let config_entity and config_architecture be the entity
-- declaration and the architecture body used to 'implement'
-- the component_instance.

-- a) The ports of the instantiated_component are eventually
```

```
--    connected to actuals (signals/ports) from the architecture
--    which contains the component_instance. These connections
--    are specified by the port_map of the component_instance.

-- b) The ports of the config_entity are associated with the
--    ports of the instantiated_component as specified by the
--    config_port_map.

-- c) From (a) and (b) it follows that the ports of config_entity
--    are eventually connected to signals/ports visible from the
--    block which contains the component_instance (parent_block).

-- That property is used at the behavioural level to specify that
-- the value of an instance of a port P of config_entity is related
-- to the value of the instance of the 'actual' connected to the port
-- P (according to (c)). If P is not connected then its (initial)
-- value is the port default value. This relationship glues together
-- two models: that of the architecture which contains the 'actual'
-- signal/port and that of the architecture corresponding to the
-- config_entity.

  config_entity: entity_declaration;
  config_architecture: architecture_body;
  config_port_map: SET OF config_component_port_association;
  config_generic_map: SET OF generic_aspect;

WHERE
  compatible_entity_and_architecture:
  config_architecture.architecture_entity = config_entity;

  valid_entity_port_configuration:
  (SIZEOF(config_entity.ports) = SIZEOF(config_port_map)) AND
  (QUERY(p <* config_entity.ports |
          SIZEOF(QUERY(assoc <* config_port_map |
                              p = assoc.entity_port)) = 1)
   = config_entity.ports);

  valid_component_port_configuration:
  (SIZEOF(SELF\component_instance.instantiated_component.ports)
   = SIZEOF(config_port_map)) AND
  (QUERY(p <* SELF\component_instance.instantiated_component.ports |
          SIZEOF(QUERY(assoc <* config_port_map |
                              p = assoc.component_port)) = 1)
   = config_entity.ports);

  -- There is a one to one association between the
  -- config_entity.ports and the instantiated_component.ports.

  some_generic_constants_are_redefined:
  QUERY(ga <* config_generic_map |
          ga.generic_const IN instantiated_component.generics)
  = config_generic_map;

  -- Some generic constants defined for the configured component
```

```
  -- instance can be redefined as far as their values are concerned.
  -- Note that a generic constant redefined at the configuration
  -- level could possibly be redefined also at the level of
  -- component instantiation. The value used at the design_schema
  -- level will be in order

  --  1) the redefined value at the configuration level
  --  2) the redefined value at the instantiation level
  --  3) the default value (taken from the generic constant
  --     declaration in the containing component).
END_ENTITY;

ENTITY config_block_port_association
       SUBTYPE OF(port_connection);
-- This is used instead a port_connection in the port_map
-- of a block.
END_ENTITY;

ENTITY config_component_port_association
       SUBTYPE OF(port_connection);
-- This is used instead a port_connection is the port_map of
-- a component instance configuration.

-- The config_component_port_association models the configuration
-- association of an entity port with a component port, according
-- to a component_instance. The configuration of the
-- component_instance is the parent_component_config.

-- By force of the association the entity port is eventually
-- connected to a signal/port from the parent_component_config
-- architecture:

-- entity_port is connected to the SELF\port_connection.locall and
-- SELF\port_connection.actual is connected to SELF\port_connection.locall
-- => entity_port is connected to SELF\port_connection.actuall

  entity_port: port;

DERIVE
  component_port: port:= SELF\port_connection.locall;

INVERSE
  parent_component_config: component_config FOR config_port_map;

WHERE
  valid_port_connection:
  SELF\port_connection IN parent_component_config\component_instance.port_map;

  -- The specialized port_connection must be containded within
  -- the port_map of the parent_configured_component.

  valid_entity_port:
  entity_port IN parent_component_config.config_entity.ports;
  -- The entity_port must belong to the entity used by the
```

```
  -- parent_component_config.

  valid_port_type:
  is_type_compatible(entity_port.object_type,
                     component_port.object_type);

  valid_port_direction:
  (is_readable(entity_port) AND is_readable(component_port)) OR
  (is_writable(entity_port) AND is_writable(component_port));
END_ENTITY;

ENTITY VHDL_block_config
       ABSTRACT SUPERTYPE OF (ONEOF(VHDL_design_unit,
                                 block_config,
                                 architecture_config))
       SUBTYPE OF(configuration_item,VHDL_block);

-- This models the configuration of a VHDL_block
-- (block or architecture).

  -- specialization of VHDL_block attributes
  SELF\VHDL_block.nested_blocks: SET OF block_config;
  SELF\VHDL_block.component_instances: SET OF component_config;

  -- additional attributes
  config_generic_map: SET OF generic_aspect;

  -- The redefinition of generic constants defined within
  -- the block to be configured.

WHERE
  some_generic_constants_are_redefined:
  QUERY(ga <* config_generic_map |
           ga.generic_const IN SELF\VHDL_block.generics)
  = config_generic_map;

  -- Some generic constants defined within the config_block
  -- can be redefined as far as their values are concerned.

   ('DESIGN_UNIT__SCHEMA.VHDL_DESIGN_UNIT' IN TYPEOF( SELF) ) OR
   ((SIZEOF(QUERY(s <* signals_and_ports |
                  'DESIGN_UNIT_SCHEMA.WIRE' IN TYPEOF(s)))= 0) AND
    (SIZEOF(QUERY(p <* processes |
                  'DESIGN_UNIT_SCHEMA.REAL_PROCESS' IN TYPEOF(p)))= 0) AND
    (SIZEOF(QUERY(c <* component_instances |
                  'DESIGN_UNIT_SCHEMA.REAL_COMPONENT' IN TYPEOF(c)))= 0) AND
    (SIZEOF(QUERY(f <* functions |
                  'DESIGN_UNIT_SCHEMA.FUNCTION_INSTANCE' IN TYPEOF(f)))= 0) AND
    (SIZEOF(QUERY(g <* generics |
                  'DESIGN_UNIT_SCHEMA.CONST' IN TYPEOF(g)))= 0));

  -- Real signals, ports, processes etc. can appear only within
  -- elaborated designs.
END_ENTITY;
```

```
ENTITY block_config
       SUPERTYPE OF(block_design)
       SUBTYPE OF(VHDL_block_config,block);

  -- specialization of block attributes
  SELF\block.nested_blocks: SET OF block_config;
  SELF\block.component_instances: SET OF component_config;
  SELF\block.port_map: config_block_port_association;
END_ENTITY;

ENTITY architecture_config
       SUPERTYPE OF(architecture_design)
       SUBTYPE OF (VHDL_block_config,architecture_body);

  -- specialization of block attributes
  SELF\architecture_body.nested_blocks: SET OF block_config;
  SELF\architecture_body.component_instances: SET OF component_config;
END_ENTITY;

-- END_SCHEMA;
```

9.7 Design unit schema

```
-- SCHEMA design_unit_schema;
-- ==============================================================
-- This schema models a simulatable design. A simulatable design
-- or a 'design_unit' is a configured architecture which operates
-- with wires (i.e. real signals and ports) which store their
-- 'waveform' updating history in time. The history of 'waveforms'
-- is structured in time according to simulation cycles.
-- ==============================================================

-- ***********************************************************
-- Here is the crucial point where the three axes of
-- the modelling space meet one another.

--       VHDL structure axis (what was so far)
--       VHDL behavioural axis (wire_states, transactions,
--            drivers and waveforms etc.)
--       Time axis (Events and simulation)
-- ***********************************************************

-- A design_unit is the specialization of a block_config
-- (** structure axis **) to conain:

--      a) wires as atomic local_objects
--      b) real_components as configured component_instances
--      c) real_processes (and real_concurrent_assignments) as
--         processes and concurrent_assignments
```

```
-- A wire is the specialization of a signal/port which stores
-- its value history (behaviour) over the simulation process
-- of the design_unit which contains the wire.
-- A wire state, an attribute of its behaviour, (** behaviour axis **)
-- is an Event (** time axis **) and therefore can be ordered in time.

-- A real_component is a configured component which
-- designates a 'real configured architecture' i.e. a design_unit

-- A real_process is a process which uses wires instead of signals
-- and ports and which contains its own transaction history
-- (** behaviour axis **).

ENTITY VHDL_design_unit
       ABSTRACT SUPERTYPE OF(ONEOF(block_design,
                                   architecture_design))
       SUBTYPE OF(VHDL_block_config);

  -- specialization of block_config attributes
  SELF\VHDL_block_config.nested_blocks: SET OF block_design;
  SELF\VHDL_block_config.parent_block: OPTIONAL VHDL_design_unit;

  SELF\VHDL_block_config.component_instances: SET OF real_component;
  SELF\VHDL_block_config.processes: SET OF real_process;

  SELF\VHDL_block_config.generics: SET OF const;
  SELF\VHDL_block_config.ports: SET OF wire;
  SELF\VHDL_block_config.guard: OPTIONAL wire;

  SELF\VHDL_block_config.signals: SET OF wire;
  SELF\VHDL_block_config.functions: SET OF function_instance;

  SELF\VHDL_block_config.parent_architecture: architecture_design;
  parent_component: OPTIONAL real_component;

DERIVE
  real_processes: SET OF real_process:= all_processes(SELF);
  wires: SET OF wire:= all_wires(SELF);

  SELF\VHDL_block_config.signals_and_ports: SET OF wire:=
  signals + ports + [guard];

  SELF\VHDL_block_config.concurrent_statements:
  SET OF real_concurrent_statement:= conv_type(processes) +
                                     conv_type(component_instances) +
                                     conv_type(nested_blocks);

  SELF\VHDL_block_config.local_objects:
  SET OF real_block_object:= conv_type(user_types) +
                             conv_type(components) +
                             conv_type(functions) +
                             conv_type(generics) +
                             conv_type(signals_and_ports);
WHERE
```

```
  inverse_parent_design_unit_for_local_objects:
  QUERY(ob <* local_objects |
           ob.parent_design_unit :=: SELF) = local_objects;
END_ENTITY;

FUNCTION all_wires
         (unit: VHDL_design_unit): SET OF wire;
-- Returns all wires contained in a VHDL_design_unit:
-- local wires + all_wires(nested_blocks) +
--                all_wires(real_components)

  LOCAL ws: SET OF wire:= unit.signals_and_ports;
        i: INTEGER;
  END_LOCAL;

  REPEAT i:= LOINDEX(unit.nested_blocks) TO
             HIINDEX(unit.nested_blocks) BY 1;
    ws:= ws + all_wires(unit.nested_blocks[i]);
  END_REPEAT;

  REPEAT i:= LOINDEX(unit.component_instances) TO
             HIINDEX(unit.component_instances) BY 1;
    ws:= ws + all_wires(unit.component_instances[i].
                        config_architecture);
  END_REPEAT;

  RETURN(ws);
END_FUNCTION;

FUNCTION all_processes
         (unit: VHDL_design_unit): SET OF real_process;
-- Returns all real processes contained in a
-- VHDL_design_unit: local processes +
--                    all_processes(nested_blocks) +
--                    all_processes(real_components)

  LOCAL ps: SET OF real_process:= unit.processes;
        i: INTEGER;
  END_LOCAL;

  REPEAT i:= LOINDEX(unit.nested_blocks) TO
             HIINDEX(unit.nested_blocks) BY 1;
    ps:= ps + all_processes(unit.nested_blocks[i]);
  END_REPEAT;

  REPEAT i:= LOINDEX(unit.component_instances) TO
             HIINDEX(unit.component_instances) BY 1;
    ps:= ps + all_processes(unit.component_instances[i].
                            config_architecture);
  END_REPEAT;

  RETURN(ps);
END_FUNCTION;
```

```
ENTITY block_design
       SUBTYPE OF(VHDL_design_unit,block_config,
                  real_concurrent_statement);

  -- Specialization of block_config (block) attributes
  SELF\block_config.nested_blocks: SET OF block_design;

  SELF\block_config.component_instances: SET OF real_component;
  SELF\block_config.processes: SET OF real_process;

  SELF\block_config.generics: SET OF const;
  SELF\block_config.ports: SET OF wire;
  SELF\block_config.guard: OPTIONAL wire;

  SELF\block_config.signals: SET OF wire;
  SELF\block_config.functions: SET OF function_instance;

  SELF\block_config.port_map: block_port_wire_connection;

DERIVE
  SELF\VHDL_design_unit.parent_component: real_component:= ?;
  -- A block cannot be used as an implementation of a component.

  SELF\block_config.parent_architecture: architecture_design:=
  SELF\real_concurrent_statement.parent_architecture;

  SELF\block_config.parent_block: VHDL_design_unit:=
  SELF\real_concurrent_statement.parent_block;

  SELF\block_config.signals_and_ports: SET OF wire:=
  signals + ports + [guard];

  SELF\block_config.concurrent_statements:
  SET OF real_concurrent_statement:= conv_type(processes) +
                                     conv_type(component_instances) +
                                     conv_type(nested_blocks);

  SELF\block_config.local_objects:
  SET OF real_block_object:= conv_type(user_types) +
                             conv_type(components) +
                             conv_type(functions) +
                             conv_type(generics) +
                             conv_type(signals_and_ports);
END_ENTITY;

ENTITY architecture_design
       SUBTYPE OF (VHDL_design_unit,architecture_config,
                   architecture_body);
  -- An architecture_design is indirectly a subtype of
  -- architecture_body since architecture_config is a
  -- subtype of architecture_body. However, DEC express is
  -- not able to derive this indirect subtyping.

  -- Specialization of architecture_config (architecture_body) attributes
```

```
  SELF\architecture_config.nested_blocks: SET OF block_design;

  SELF\architecture_config.component_instances: SET OF real_component;
  SELF\architecture_config.processes: SET OF real_process;

  SELF\architecture_config.generics: SET OF const;
  SELF\architecture_config.ports: SET OF wire;
  SELF\architecture_config.guard: OPTIONAL wire;

  SELF\architecture_config.signals: SET OF wire;
  SELF\architecture_config.functions: SET OF function_instance;

DERIVE
  SELF\VHDL_design_unit.parent_component: real_component:=
  get_parent_component(parent_components);

  SELF\architecture_config.parent_architecture: architecture_design:= SELF;

  SELF\architecture_config.parent_block: VHDL_design_unit:= ?;

  SELF\architecture_config.signals_and_ports: SET OF wire:=
  signals + ports + [guard];

  SELF\architecture_config.concurrent_statements:
  SET OF real_concurrent_statement:= conv_type(processes) +
                                     conv_type(component_instances) +
                                     conv_type(nested_blocks);

  SELF\architecture_config.local_objects:
  SET OF real_block_object:= conv_type(user_types) +
                             conv_type(components) +
                             conv_type(functions) +
                             conv_type(generics) +
                             conv_type(signals_and_ports);
INVERSE
  parent_components:
  SET [0 : 1] OF real_component FOR config_architecture;
END_ENTITY;

FUNCTION get_parent_component
         (cs: SET OF real_component): real_component;

  IF cs <> [] THEN RETURN(cs[LOINDEX(cs)]);
            ELSE RETURN(?);
  END_IF;
END_FUNCTION;

ENTITY real_block_object
       ABSTRACT SUPERTYPE OF
                (ONEOF(real_atomic_object,component_template,
                       function_instance,user_type_instance))
       SUBTYPE OF(block_object);

-- In order to specialize a config_block all its attributes
```

```
-- must be correctly specialized for the correct inheritance of
-- all structural properties defined at the level of structure_schema.

  parent_design_unit: VHDL_design_unit;

DERIVE
  parent_architecture_design: architecture_design:=
  parent_design_unit.parent_architecture;

WHERE SELF IN parent_design_unit.local_objects;
END_ENTITY;

ENTITY component_template SUBTYPE OF(real_block_object,component);
-- It is as at the structure level but, in addition,
-- it is considered to be a real_block_object.
END_ENTITY;

ENTITY function_instance
       ABSTRACT SUPERTYPE OF (ONEOF(resolution_function_instance,
                                    conversion_function_instance))
       SUBTYPE OF(real_block_object,VHDL_function);
-- It is as at the structure level but, in addition,
-- it can be an instance of real_block_object.
END_ENTITY;

ENTITY resolution_function_instance
       SUBTYPE OF(function_instance,resolution_function);
END_ENTITY;

ENTITY conversion_function_instance
       SUBTYPE OF(function_instance,resolution_function);
END_ENTITY;

ENTITY user_type_instance SUBTYPE OF(real_block_object,user_type);
-- It is as at the structure level but, in addition,
-- it can be an instance of real_block_object.
END_ENTITY;

ENTITY real_atomic_object
       ABSTRACT SUPERTYPE OF(ONEOF(wire,const))
       SUBTYPE OF(real_block_object,atomic_object);
END_ENTITY;

ENTITY wire SUBTYPE OF(real_atomic_object,signal_or_port,signal,port);

  -- Specialization of attributes
  SELF\signal_or_port.sources: SET OF wire_source;
  SELF\signal_or_port.resolved_by: OPTIONAL resolution_function_instance;

  -- Additional attributes
  behaviour: wire_history;

WHERE
  same_parent_architecture_design_for_functions(SELF,resolved_by);
```

```
  -- The wire and its resolution function must be in the same
  -- architecture_design unit.
END_ENTITY;

FUNCTION same_parent_architecture_design_for_functions
         (w: wire; f: function_instance): LOGICAL;

  IF NOT EXISTS(f) THEN RETURN(TRUE); END_IF;
  RETURN(f\real_block_object.parent_architecture_design :=:
         w\real_block_object.parent_architecture_design);
END_FUNCTION;

ENTITY wire_source
       ABSTRACT SUPERTYPE OF (ONEOF(wire_connection,wire_assignment))
       SUBTYPE OF (signal_or_port_source);

DERIVE
  SELF\signal_or_port_source.target: wire:= wire_target;

INVERSE
  wire_target: wire FOR sources;
END_ENTITY;

ENTITY wire_connection
       ABSTRACT SUPERTYPE OF(ONEOF (component_port_wire_connection,
                                    block_port_wire_connection))
       SUBTYPE OF(wire_source,port_connection);

  -- Specializations of port_connection attributes
  SELF\port_connection.locall: wire;
  SELF\port_connection.actual: wire;
  SELF\port_connection.source: wire;
  SELF\port_connection.converted_by:
      OPTIONAL conversion_function_instance;

DERIVE
  SELF\port_connection.target: wire:= SELF\wire_source.target;

WHERE
  source\real_block_object.parent_architecture_design :=:
  SELF\wire_source.target\real_block_object.parent_architecture_design;
  -- The connection source and target must be in the same
  -- architecture_design unit.

  same_parent_architecture_design_for_functions(source,converted_by);
  -- The source wire and its type-conversion function must
  -- be in the same architecture_design unit.
END_ENTITY;

ENTITY block_port_wire_connection
       SUBTYPE OF(wire_connection,config_block_port_association);
  -- This entity stands for the connection of a wire from
  -- the current design unit to a wire from another design
  -- unit.
```

```
  SELF\config_block_port_association.locall: wire;
  SELF\config_block_port_association.actual: wire;
  SELF\config_block_port_association.source: wire;
  SELF\config_block_port_association.converted_by:
      OPTIONAL conversion_function_instance;

DERIVE
  SELF\config_block_port_association.target:
  wire:= SELF\wire_connection.target;

INVERSE
  connection_parent_block: block_design FOR port_map;
END_ENTITY;

ENTITY component_port_wire_connection
       SUBTYPE OF(wire_connection,config_component_port_association);
  -- This entity stands for the connection of a wire from
  -- the current design unit to a interface wire from a real
  -- component.

  SELF\config_component_port_association.locall: wire;
  SELF\config_component_port_association.actual: wire;
  SELF\config_component_port_association.source: wire;
  SELF\config_component_port_association.converted_by:
      OPTIONAL conversion_function_instance;

  SELF\config_component_port_association.entity_port: wire;

DERIVE
  SELF\config_component_port_association.target:
  wire:= SELF\wire_connection.target;

INVERSE
  connection_parent_component: real_component FOR config_port_map;

WHERE
  QUERY(ls <* locall.behaviour |
        QUERY(es <* entity_port.behaviour |
                 (ls.sim_cycle :<>: es.sim_cycle) OR
                 (ls.current_value :=: es.current_value))
        = entity_port.behaviour)
  = locall.behaviour;

  -- The local of a real connection and the entity_port
  -- to which it is connected have the same values in
  -- the same simulation cycle.
END_ENTITY;

ENTITY wire_assignment
       SUBTYPE OF(wire_source,signal_assignment);

   -- Specialization of signal_assignment attributes
   SELF\signal_assignment.assignment_signals: SET OF wire;
```

```
INVERSE
   parent_process: real_process FOR assignments;
END_ENTITY;

ENTITY const SUBTYPE OF(real_atomic_object,generic_constant);
DERIVE
  const_value: VHDL_value:=
  real_generic_constant_value(SELF);
END_ENTITY;

FUNCTION real_generic_constant_value
         (gc: const): VHDL_value;
(*
  I) If the parent_unit P is an architecture_design which implements
  the configured component instance CI then

   a) lookup for gc in the config_generic_map of the configuration
      of the component instance CI;

   b) lookup for gc in the generic_map of the component instance CI

   c) lookup for gc in the config_generic_map of the parent_unit P.

   II) If the parent_unit P is the main architecture_design or
   if it is a block_design (it has no parent_component) then

   c) lookup for gc in the config_generic_map of the parent_unit P.
*)
  LOCAL parent_unit: VHDL_design_unit:=
                     gc\real_atomic_object.parent_design_unit;

        parent_component: real_component:=
                          parent_unit.parent_component;
        v: VHDL_value;
  END_LOCAL;

  IF parent_unit.parent_component <> ?
  THEN
       v:= lookup_generic_constant
           (gc\generic_constant,
            parent_component\component_config.config_generic_map);
       IF v <> ? THEN RETURN(v); END_IF;

       v:= lookup_generic_constant
           (gc\generic_constant,
            parent_component\component_config.generic_map);
       IF v <> ? THEN RETURN(v); END_IF;
   END_IF;

   v:= lookup_generic_constant
       (gc\generic_constant,
        parent_unit\VHDL_block_config.config_generic_map);
   IF v <> ? THEN RETURN(v); END_IF;
```

```
   RETURN(gc\generic_constant.default_value);
END_FUNCTION;

FUNCTION lookup_generic_constant
         (gc:  generic_constant;
          map: SET OF generic_aspect): VHDL_value;
-- Search for a generic constant gc in an association map.

  LOCAL i: INTEGER; END_LOCAL;
  REPEAT i:= LOINDEX(map) TO HIINDEX(map) BY 1;
    IF gc :=: map[i].generic_const
    THEN RETURN(map[i].assoc_value);
    END_IF;
  END_REPEAT;
  RETURN(?);
END_FUNCTION;

ENTITY real_concurrent_statement
       ABSTRACT SUPERTYPE OF (ONEOF(real_component,real_process,block_design))
       SUBTYPE OF(concurrent_statement);

  SELF\concurrent_statement.parent_block: VHDL_design_unit;

DERIVE
  SELF\concurrent_statement.parent_architecture:
  architecture_design:= parent_block.parent_architecture;
END_ENTITY;

ENTITY real_component
       SUBTYPE OF(real_concurrent_statement,component_config);
  SELF\component_config.config_architecture: architecture_design;
  SELF\component_config.config_port_map:
      SET OF component_port_wire_connection;

WHERE
  QUERY(con <* config_port_map |
        con.locall\real_block_object.parent_architecture_design :=:
        SELF\real_concurrent_statement.parent_architecture)
  = config_port_map;

  -- The local wire of each connection of the component
  -- and the component must be in the same architecture_design.
END_ENTITY;

ENTITY real_process
       ABSTRACT SUPERTYPE OF(ONEOF(real_process_statement,
                                   concurrent_wire_assignment))
       SUBTYPE OF(real_concurrent_statement,process);

  -- Specialization of process direct attributes.
  SELF\process.consulted_signals: SET OF wire;

  SELF\process.assignments: SET OF wire_assignment;
```

```
  SELF\process.signal_waits: SET OF wire_wait;
  SELF\process.delay_waits: SET OF delay_wait;
  SELF\process.condition_waits: SET OF condition_wait;

  behaviour: SET OF real_process_history;

DERIVE
  -- Specialized derived attributes (type specialization only).
  SELF\process.guard_signal: wire:=
  SELF\real_concurrent_statement.parent_block.guard;

  SELF\process.sensitivity_signals: SET OF wire:=
  wires_from_signal_waits(signal_waits);

  SELF\process.assign_signals: SET OF wire:=
  assignment_wires(assignments);
  -- Wires used in the right part of wire assignments.

  SELF\process.controlled_signals: SET OF wire:=
  assignment_target_wires(assignments);

  condition_signals: SET OF wire:=
  wires_from_condition_waits(condition_waits);

  signals: SET OF wire:= controlled_signals +
                         sensitivity_signals +
                         consulted_signals;
  -- The set of all wires used within the process.

WHERE
  QUERY(w <* (consulted_signals + controlled_signals) |
        w\real_block_object.parent_architecture_design :=:
        SELF\real_concurrent_statement.parent_architecture)
  = (consulted_signals + controlled_signals);

  -- All of the signals used by the process and the
  -- process must be in the same architecture_design unit.
END_ENTITY;

ENTITY wire_wait SUBTYPE OF(signal_wait_statement);
  SELF\signal_wait_statement.sensitivity_signals: SET [1 : ?] OF wire;

INVERSE
  parent_process: real_process FOR signal_waits;
END_ENTITY;

ENTITY condition_wait SUBTYPE OF(condition_wait_statement);
  SELF\condition_wait_statement.condition_signals: SET [1 : ?] OF wire;

  behaviour: condition_wait_history;
  -- The set of all values tken by the condition expression
  -- during the simulation process.

INVERSE
```

```
  parent_process: real_process FOR condition_waits;
END_ENTITY;

ENTITY delay_wait SUBTYPE OF(delay_wait_statement);
INVERSE
  parent_process: real_process FOR delay_waits;
END_ENTITY;

ENTITY concurrent_wire_assignment
       SUBTYPE OF(real_process,concurrent_assignment);
END_ENTITY;

ENTITY real_process_statement
       SUBTYPE OF(real_process,process_statement);
END_ENTITY;

FUNCTION assignment_target_wires
         (assign: SET OF wire_assignment): SET OF wire;

  LOCAL i: INTEGER:= LOINDEX(assign);
        s: SET OF wire:= [];
  END_LOCAL;

  REPEAT i:= LOINDEX(assign) TO HIINDEX(assign) BY 1;
         s := s + [assign[i].target];
  END_REPEAT;
  RETURN(s);
END_FUNCTION;

FUNCTION assignment_wires
         (assign: SET OF wire_assignment): SET OF wire;

  LOCAL i: INTEGER:= LOINDEX(assign);
        s: SET OF wire:= [];
  END_LOCAL;

  REPEAT i:= LOINDEX(assign) TO HIINDEX(assign) BY 1;
         s := s + assign[i].assignment_signals;
  END_REPEAT;
  RETURN(s);
END_FUNCTION;

FUNCTION wires_from_signal_waits
        (w: SET OF wire_wait): SET OF wire;

  LOCAL i: INTEGER;
        s: SET OF wire:= [];
  END_LOCAL;

  REPEAT i:= LOINDEX(w) TO HIINDEX(w) BY 1;
      s:= s + w[i].sensitivity_signals;
  END_REPEAT;
  RETURN(s);
END_FUNCTION;
```

```
FUNCTION wires_from_condition_waits
         (w: SET OF condition_wait): SET OF wire;

  LOCAL i: INTEGER;
        s: SET OF wire:= [];
  END_LOCAL;

  REPEAT i:= LOINDEX(w) TO HIINDEX(w) BY 1;
      s:= s + w[i].condition_signals;
  END_REPEAT;
  RETURN(s);
END_FUNCTION;

-- END_SCHEMA;
```

9.8 Behaviour schema

```
-- SCHEMA behaviour_schema;
-- ======================================================
-- This is the model of causal relationships between
--  waveform states and process transactions (assignment,
-- wait). These relationships are anchored in time, by
-- means of simulation cycles.
-- ======================================================

ENTITY wire_state;
-- Represents the state of a wire in a simulation_cycle.

  drivers: SET OF driver;
  resolved_by: OPTIONAL resolution_computation;

  previous_state: OPTIONAL wire_state;
  -- The state of the parent_wire in the previous
  -- simulation cycle, if that exists.

  -- The current_value is the value computed either by
  -- the resolution computation (if the parent wire is
  -- resolved) or the current value of the wire source
  -- if the parent wire is not resolved. It is the value
  -- of the parent_wire in the sim_cycle.

  -- If the current_value != old_value (previous sim_cycle)
  -- then an event happened on the wire (it really changed
  -- its value). The old_value on the next simulation cycle
  -- is the current_value on the current sim. cycle.

  -- Note that the type conformance between the params.
  -- of the resolution_computation and the drivers is
  -- guaranteed by the constraints at the structure
  -- level (drivers are indirectly signals_or_ports,
  -- resolution_computation is a resolution_function).
```

```
DERIVE
  current_value: VHDL_value:= compute_wire_value(SELF);
  signal_event: LOGICAL:= compute_signal_event(SELF);
  parent_wire: wire:= parent_wire_history.parent_wire;

INVERSE
  parent_wire_history: wire_history FOR wire_states;
  sim_cycle: simulation_cycle FOR wire_states;

  satisfied_waits: SET OF signal_wait_transaction FOR outcome;

WHERE
  current_value.object_type = parent_wire.object_type;
  -- The current_value is type compatible with the type of
  -- the parent_wire.

  (SIZEOF(drivers) = SIZEOF(parent_wire.sources)) AND
  (QUERY(d <* drivers | d.parent_wire_source IN parent_wire.sources)
   = drivers);
  -- The drivers are compatible with the parent_wire.
  -- Triangularization constraint driver --> wire_source
  --                              driver --> parent_wire

  (parent_wire.is_resolved AND EXISTS(resolved_by)) OR
  (NOT parent_wire.is_resolved AND NOT EXISTS(resolved_by));
  -- The resolution_computation exists only if the parent_wire
  -- is resolved.

  previous_state_is_valid: check_wire_previous_state(SELF);

  -- If the wire_state occurs at the beginning of simulation
  -- then it has no previous_state. Else there is a previous
  -- state of the same wire such that SELF is next to that.
END_ENTITY;

ENTITY resolution_computation;
-- The resolution_computation is an execution instance
-- the resolution_function of the parent_wire.

  resolved_value: VHDL_value;
  -- This is the result of the resolution_computation.

DERIVE
  parameters: SET OF driver:= parent_wire_state.drivers;
  parent_wire: wire:= parent_wire_state.parent_wire;

  parent_resolution_function: resolution_function_instance:=
  parent_wire.resolved_by;

INVERSE
  parent_wire_state: wire_state FOR resolved_by;

WHERE
```

```
  is_type_compatible(resolved_value.object_type,
                     parent_resolution_function.object_type);
END_ENTITY;

FUNCTION check_wire_previous_state(ws: wire_state): LOGICAL;
  IF is_simulation_start_cycle(ws.sim_cycle)
  THEN RETURN(NOT EXISTS(ws.previous_state));
  ELSE IF NOT EXISTS(ws.previous_state)
       THEN RETURN(FALSE);
       ELSE RETURN(is_next(ws.sim_cycle\Event,
                           ws.previous_state.sim_cycle\Event) AND
                  (ws.parent_wire :=: ws.previous_state.parent_wire));
       END_IF;
  END_IF;
END_FUNCTION;

FUNCTION compute_wire_value
         (ws: wire_state): VHDL_value;

  IF is_simulation_start_cycle(ws.sim_cycle)
  THEN RETURN(ws.parent_wire.default_value);
  END_IF;

  IF ws.parent_wire.is_resolved
  THEN RETURN (ws.resolved_by.resolved_value);
  END_IF;

  IF ws.drivers <> []
  THEN RETURN (ws.drivers[LOINDEX(ws.drivers)].driver_current_value);
  ELSE RETURN (ws.parent_wire.default_value);
  END_IF;
END_FUNCTION;

FUNCTION compute_signal_event(ws: wire_state): LOGICAL;
  IF is_simulation_start_cycle(ws.sim_cycle)
  THEN RETURN(TRUE);
  ELSE RETURN(ws.previous_state.current_value <> ws.current_value);
  END_IF;
END_FUNCTION;

ENTITY driver
       ABSTRACT SUPERTYPE OF(ONEOF(connection_driver,assign_driver));
-- This is the instantaneous state of a wire_source. It controls
-- the current_value of the target wire_state.

  parent_wire_source: wire_source;
  driver_current_value: VHDL_value;
  previous_driver: OPTIONAL driver;

DERIVE
  sim_cycle: simulation_cycle:= parent_wire_state.sim_cycle;

INVERSE
  parent_wire_state: wire_state FOR drivers;
```

```
WHERE
  previous_driver_is_valid: check_previous_driver(SELF);

  -- If the driver occurs at the beginning of simulation
  -- then it has no previous_driver. Else there is a previous
  -- driver for the same wire such that SELF is next to that.
END_ENTITY;

FUNCTION check_previous_driver(d: driver): LOGICAL;
  IF is_simulation_start_cycle(d.sim_cycle)
  THEN RETURN(NOT EXISTS(d.previous_driver));
  ELSE IF NOT EXISTS(d.previous_driver)
       THEN RETURN(FALSE);
       ELSE RETURN((d.parent_wire_source :=:
                    d.previous_driver.parent_wire_source) AND
                   (is_next(d.sim_cycle\Event,
                            d.previous_driver.sim_cycle\Event)));
            -- Same parent_wire_source => same parent_wire.
       END_IF;
  END_IF;
END_FUNCTION;

FUNCTION is_assign_driver(x: driver): LOGICAL;
  RETURN(is_type(x,'BEHAVIOUR_SCHEMA.ASSIGN_DRIVER'));
END_FUNCTION;

FUNCTION is_connection_driver(x: driver): LOGICAL;
  RETURN(is_type(x,'BEHAVIOUR_SCHEMA.CONNECTION_DRIVER'));
END_FUNCTION;

ENTITY connection_driver SUBTYPE OF(driver);
-- This entity models the instantaneous state of connection
-- between two wires. Note that these wires belong to different
-- design_units (as specified in the wire_connection).

  parent_wire_connection: wire_connection;
  connection_source: wire_state;
  type_conversion: OPTIONAL type_conversion_computation;

DERIVE
  SELF\driver.driver_current_value: VHDL_value:=
  connection_driver_value(SELF);

  SELF\driver.parent_wire_source: wire_connection:=
  parent_wire_connection;

  connection_target: wire_state:= SELF\driver.parent_wire_state;

WHERE
  connection_source.sim_cycle :=: connection_target.sim_cycle;

  -- The states of the connected wires are concurrent
  -- ie. they occur in the same simulation cycle
```

```
  connection_source.parent_wire :=: parent_wire_connection.source;
  connection_target.parent_wire :=: parent_wire_connection.target;

  -- The driver is IN the set of wire_sources of the
  -- parent_wire_state.parent_wire, hence the connection_target
  -- is the parent_wire_state.

  (parent_wire_connection.is_converted AND EXISTS(type_conversion)) OR
  (NOT parent_wire_connection.is_converted AND NOT EXISTS(type_conversion));
  -- The type_conversion computation exists only if the parent
  -- wire_connection is type converted.
END_ENTITY;

ENTITY type_conversion_computation;
-- This is an execution instance of a conversion_function.

  converted_value: VHDL_value;
  -- This is the result of the type conversion_computation.

DERIVE
  parameter: wire_state:=
  parent_connection_driver.connection_source;

  parent_wire_connection: wire_connection:=
  parent_connection_driver.parent_wire_connection;

  parent_conversion_function: conversion_function_instance:=
  parent_wire_connection.converted_by;

INVERSE
  parent_connection_driver: connection_driver FOR type_conversion;

WHERE
  is_type_compatible(converted_value.object_type,
                     parent_conversion_function.object_type);
END_ENTITY;


FUNCTION connection_driver_value(cd: connection_driver): VHDL_value;
-- Computes the value of a connection_driver.
-- If the connection_source is type converted then the
-- result is the value of the conversion computation, else
-- it is the current value of the connection_source.

  IF cd.parent_wire_connection.is_converted
  THEN RETURN(cd.type_conversion.converted_value);
  ELSE RETURN(cd.connection_source.current_value);
  END_IF;
END_FUNCTION;

ENTITY assign_driver SUBTYPE OF(driver);

-- This models the driving of a wire by the signal
```

```
-- assignments from a process (or concurrent assignment).
-- The current waveform + the set of projected values
-- generate a new waveform, which is equal to the wire
-- state current waveform in the next simulation cycle.

  parent_wire_assignment: wire_assignment;
  current_assign: OPTIONAL assign_transaction;
  -- The current assign_transaction generated by the
  -- wire_assignment. It may not exist if the wire_assignment
  -- parent process is blocked by force of a wait_transaction
  -- not yet satisfied.

DERIVE
  SELF\driver.parent_wire_source:
  wire_assignment:= parent_wire_assignment;

  assign_inhibit: LOGICAL:= compute_assign_inhibit(SELF);
  -- It is used to block the waveform transform of a driver
  -- corresponding to a guarded (concurrent) signal assignment
  -- with a false guard or to a blocked assignment (contained
  -- within a blocked process).

  current_waveform:
  waveform:= assign_driver_current_waveform(SELF);

  -- It is the current waveform of the driver from the previous
  -- simulation cycle, if any. At the beginning of simulation
  -- the waveform contains the dafault value of the driver parent
  -- wire.

  SELF\assign_driver.new_waveform: waveform:=
  waveform_transform(current_waveform,
                     current_assign,
                     SELF\driver.sim_cycle,
                     assign_inhibit);

  -- This the new waveform (i.e. the current waveform for
  -- the next simulation cycle.)

  SELF\driver.driver_current_value: VHDL_value:=
  assign_driver_value(SELF);

  driver_old_value: VHDL_value:= assign_driver_old_value(SELF);

  -- It stores the value the driver had in the previous
  -- simulation cycle. If the current_waveform does not
  -- contain a value scheduled for the current time
  -- (sim. cycle) then the old value becomes the current
  -- value of the driver.

WHERE
  check_current_assign(SELF);
  -- The current_assign and the driver must be in the same
  -- simulation cycle and the wire_assignment which generated
```

```
  -- the current_assign transaction must be the wire_assignment
  -- the driver is associated with.

  parent_wire_assignment.target :=:
  SELF\driver.parent_wire_state.parent_wire;

  -- This is for clarity. Since the assign_driver is IN
  -- the parent_wire_state.parent_wire wire_sources, the
  -- parent_wire is the target of the assignment.
END_ENTITY;

FUNCTION check_current_assign(d: assign_driver): LOGICAL;
-- This is a typical example of a simple function which encodes
-- a logical implication. Shouldn't EXPRESS do it directly ?

  IF NOT EXISTS(d.current_assign) THEN RETURN(TRUE); END_IF;
  RETURN((d.current_assign\transaction.sim_cycle :=:
          d\driver.sim_cycle) AND
         (d.current_assign.parent_wire_assignment :=:
          d.parent_wire_assignment));
END_FUNCTION;

FUNCTION assign_driver_old_value
         (d: assign_driver): VHDL_value;

  IF NOT EXISTS(d\driver.previous_driver)
  THEN RETURN(d\driver.parent_wire_state.parent_wire.default_value);
  ELSE RETURN(d\driver.previous_driver.driver_current_value);
  END_IF;
END_FUNCTION;

FUNCTION assign_driver_current_waveform
         (d: assign_driver): waveform;
  LOCAL
     initial_projected_value: projected_value:=
     projected_value(d\driver.parent_wire_state.
                       parent_wire.default_value,
                     d\driver.sim_cycle.
                       parent_simulation.simulation_start_time);
  END_LOCAL;

  IF NOT EXISTS(d\driver.previous_driver)
  THEN RETURN(waveform([initial_projected_value]));
  ELSE RETURN(d\driver.previous_driver\assign_driver.new_waveform);
  END_IF;
END_FUNCTION;

FUNCTION assign_driver_value
         (d: assign_driver): VHDL_value;

  -- It computes the current value of the assign_driver d.

  LOCAL wev: projected_value:=
            d.current_waveform.earliest_value;
```

```
  END_LOCAL;

  IF (wev = ?) OR d.assign_inhibit
  THEN RETURN(d.driver_old_value);
  END_IF;
  -- The current_waveform is empty (as it always is
  -- at the start of the simulation)

  IF wev.projected_time > d\driver.sim_cycle\Event.event_time
  THEN RETURN(d.driver_old_value);
  END_IF;
  -- The earliest_value of the current_waveform
  -- will eventually be scheduled in the future.

  RETURN(wev.new_value);
  -- The earliest value is concurrent with the driver and
  -- implicitly with the wire state (it occurs "now")
END_FUNCTION;

FUNCTION compute_assign_inhibit
         (d: assign_driver): LOGICAL;

  IF NOT EXISTS(d.current_assign) THEN RETURN(TRUE); END_IF;
  IF d.current_assign.guarded_assignment
  THEN
       RETURN(d.current_assign.guard_wire_state.current_value = VHDL_true);
  ELSE RETURN(FALSE);
  END_IF;
END_FUNCTION;

FUNCTION waveform_transform
         (w: waveform;
          at: assign_transaction;
          sim_cycle: simulation_cycle;
          assign_inhibit: LOGICAL): waveform;

  -- Transforms the waveform w according to the set
  -- of projected_values contained wihtin the assign
  -- effects and the corresponding assign mode
  -- (transport or inertial).

  -- An assign_effect represents the effect of executing
  -- a single signal assignment.

  LOCAL i: INTEGER;
        ae: LIST OF assign_effect;
  END_LOCAL;

  -- Eliminate from w the value scheduled for the current
  -- simulation_cycle.event_time, if any.
  w:= delete_projected_value(w,sim_cycle\Event.event_time);

  IF assign_inhibit THEN RETURN(w); END_IF;
```

```
  ae:= at.assign_effects;
  REPEAT i:= LOINDEX(ae) TO HIINDEX(ae) BY 1;
  -- Transform the waveform according to the projected
  -- values generated by at.assign_effects[i].
     w:= one_step_waveform_transform
         (w,ae[i].assign_values,ae[i].assign_mode);
  END_REPEAT;
  RETURN(w);
END_FUNCTION;

FUNCTION delete_projected_value
         (w: waveform; t: Time_value): waveform;

  -- Eliminates from w the projected values with
  -- projected_time = t.

  LOCAL pv: SET OF projected_value:= [];
        i: INTEGER;
  END_LOCAL;

  REPEAT i:= LOINDEX(pv) TO HIINDEX(pv) BY 1;
    IF w.waveform_values[i].projected_time <> t
    THEN pv:= pv + [w.waveform_values[i]];
    END_IF;
  END_REPEAT;
  RETURN(waveform(pv));
END_FUNCTION;

FUNCTION one_step_waveform_transform
         (w: waveform;
          pv: LIST OF projected_value;
          assign_mode: transport_or_inertial): waveform;

  -- This function is not a smart transcription of the IEEE
  -- Std 1076-1987 algorithm. Instead it is intended to be
  -- as close as possible to that algorithm.

  LOCAL i: INTEGER;
        new: SET OF projected_value:= list_to_set(pv);
        old: SET OF projected_value:= w.waveform_values;
        marked: SET OF projected_value;
        val: projected_value;

        min_time: Time_value:= pv[LOINDEX(pv)].projected_time;
        -- pv is sorted in ascending order on projected_time
  END_LOCAL;

  -- Purge the old set of projected values by eliminating
  -- all projected values scheduled after the earliest
  -- assigned value.

  REPEAT j:= LOINDEX(old) TO HIINDEX(old) BY 1;
    IF old[i].projected_time <= min_time
    THEN new:= new + [old[i]];
```

```
    END_IF;
  END_REPEAT;

  IF assign_mode = transport
  THEN RETURN(waveform(new));
  END_IF;

  -- Inertial assignment. Uses the algorithm described
  -- in the IEEE Std 1076-1987 page 8-5.

  old:= new;              -- The 'transport' processed set of
                          -- projected values.
  new:= list_to_set(pv);  -- The extended set of marked values.
  marked:= [];            -- The old set of marked values.

  -- Extend the marked value set as much as possible
  -- with value-equal immediate predecessors from the
  -- old set.

  REPEAT WHILE (new <> marked);
    marked:= new;

    REPEAT i:= LOINDEX(marked) TO HIINDEX(marked) BY 1;
      val:= immediate_predecessor(old,marked[i]);
      IF val <> ? -- If val exists
      THEN
           IF val.new_value = marked[i].new_value
           THEN new:= new + [val];
           END_IF;
      END_IF;
    END_REPEAT;
  END_REPEAT;

  RETURN(waveform(new));
END_FUNCTION;

FUNCTION list_to_set(l: LIST OF GENERIC): SET OF GENERIC;
  LOCAL i: INTEGER;
        s: SET OF GENERIC:= [];
  END_LOCAL;

  REPEAT i:= LOINDEX(l) TO HIINDEX(l) BY 1;
    s:= s + [l[i]];
  END_REPEAT;
  RETURN(s);
END_FUNCTION;

FUNCTION immediate_predecessor
         (pv: SET OF projected_value;
          v: projected_value): projected_value;

-- Returns the immediate time predecessor of v from
-- the set pv, or ? if there is no predecessor for v.
```

```
  LOCAL i: INTEGER;
        pred: projected_value:= ?;
  END_LOCAL;

  REPEAT i:= LOINDEX(pv) TO HIINDEX(pv) BY 1;
    IF pv[i].projected_time < v.projected_time
    THEN
        IF pred = ?
        THEN pred:= pv[i];
        ELSE
            IF pred.projected_time < pv[i].projected_time
            THEN pred:= pv[i];
            END_IF;
        END_IF;
     END_IF;
  END_REPEAT;
  RETURN(pred);
END_FUNCTION;

ENTITY waveform;
  waveform_values: SET OF projected_value;

  -- Note that a waveform is in a sim_cycle (since it
  -- is a simulation_value).
  -- A waveform has a value (the earliest projected value).

DERIVE
  earliest_value:
  projected_value:= get_waveform_earliest_value(waveform_values);

  waveform_scheduled_time:
  Time_value:= earliest_value.projected_time;

  -- This is the time the waveform will eventually
  -- be sampled during the simulation process.

INVERSE
  parent_driver: assign_driver FOR current_waveform;

WHERE
  QUERY(pv <* waveform_values |
            pv.projected_time >=
            parent_driver\driver.sim_cycle\Event.event_time)
  = waveform_values;

  -- The projected values from the waveform must be
  -- scheduled in the future. Since delta delays are
  -- possible, time_value equality is permitted.
END_ENTITY;

FUNCTION get_waveform_earliest_value
         (wf: SET OF projected_value): projected_value;
-- It searches wf set for the minimum time waveform value.
```

```
  LOCAL i: INTEGER;
        result: projected_value;
  END_LOCAL;

  IF wf = [] THEN RETURN (?); END_IF;
  result:= wf[LOINDEX(wf)];

  REPEAT i:= LOINDEX(wf) TO HIINDEX(wf) BY 1;
    IF result.projected_time > wf[i].projected_time
    THEN result:= wf[i];
    END_IF;
  END_REPEAT;
  RETURN (result);
END_FUNCTION;

ENTITY transaction ABSTRACT SUPERTYPE OF
      (ONEOF (wait_transaction,assign_transaction));

-- Models the effect of executing a (synthetic) signal
-- assignment or a wait statement.

DERIVE
  parent_real_process: real_process:=
  parent_process_history.parent_real_process;

INVERSE
  sim_cycle: simulation_cycle FOR transactions;
  parent_process_history:
  real_process_history FOR transactions;
END_ENTITY;

ENTITY wait_transaction
       ABSTRACT SUPERTYPE OF
               (ONEOF (delay_wait_transaction,
               signal_wait_transaction,
                       condition_wait_transaction))
       SUBTYPE OF (transaction);

  next_wait: OPTIONAL wait_transaction;
  -- Executing a wait statement in a process leads to
  -- the execution of another wait statement in the
  -- same process in a subsequent simulation cycle, not
  -- necessarily the "next" one. Note that a signal_wait
  -- transaction may become blocked and therefore may
  -- not have a next_wait.

  outcome_cycle: OPTIONAL simulation_cycle;
  -- The cycle in which the wait_transaction is satisfied.

INVERSE
  effects: SET OF assign_transaction FOR wait_cause;
  -- The wait_transaction is always the cause of executing
  -- a process (and therefore signal assignments).
  -- The assign_transaction, which models the action
```

```
  -- of executing a signal assignment, can have as a
  -- 'wait_cause' a wait_transaction.

  wait_cause: SET [0 : 1] OF wait_transaction FOR next_wait;

WHERE
  (is_simulation_start_cycle(sim_cycle) AND (wait_cause = [])) OR
  (NOT is_simulation_start_cycle(sim_cycle) AND (wait_cause <> []));

  -- The cause of executing a wait statement is another wait,
  -- except for the start of the simulation.

  (NOT EXISTS(outcome_cycle) AND NOT EXISTS(next_wait)) OR
  (EXISTS(outcome_cycle) AND EXISTS(next_wait));

  -- If the wait is satisfied then the parent process
  -- resumes its execution until a next_wait is executed.
  -- If the wait is not satisfied (it can happen for a
  -- signal_wait) then the parent process is blocked
  -- for ever.

  (NOT EXISTS(outcome_cycle) AND (effects = [])) OR
  EXISTS(outcome_cycle);

  -- If the wait transaction is satisfied then some
  -- signal assignments from the parent process, if
  -- any, can be eventually executed. Note that there
  -- can be no effects.

  QUERY(at <* effects |
            at.parent_real_process :=: parent_real_process)
  = effects;

  -- The signal assignments executed as result of
  -- satisfying this wait_transaction belong to the
  -- parent process of the wait transaction.

  wait_transaction_effects_scheduling(SELF);

  -- The next_wait transaction and the assignment_transactions
  -- on the effects role of this wait belong to the same
  -- process_statement, are concurrent and occur in the
  -- outcome_cycle.
END_ENTITY;

FUNCTION wait_transaction_effects_scheduling
         (w: wait_transaction): LOGICAL;

  IF NOT EXISTS(w.outcome_cycle) THEN RETURN(TRUE); END_IF;
  RETURN(
         is_after(w.outcome_cycle\Event,w.sim_cycle\Event) AND
         (QUERY(at <* w.effects | at.sim_cycle :=: w.outcome_cycle)
          = w.effects) AND
         (w.next_wait.parent_real_process :=: w.parent_real_process) AND
```

```
        (w.next_wait.sim_cycle :=: w.outcome_cycle));

  -- The outcome_cycle is used to synchronize both the
  -- future signal assignments and the next wait executed
  -- in the parent process of w.
END_FUNCTION;

ENTITY signal_wait_transaction
       SUBTYPE OF (wait_transaction);
-- Models a signal wait statement, which suspends the
-- parent process, waiting for a change in the value
-- of some signals.

  parent_wire_wait: wire_wait;

  outcome: SET OF wire_state;
  -- The wire states on which the signal_wait depends.
  -- The set is [] if the transaction is not satisfied.

DERIVE
  SELF\wait_transaction.outcome_cycle: simulation_cycle:=
  compute_signal_wait_outcome_cycle(outcome);

WHERE
  parent_wire_wait.parent_process :=: parent_real_process;

  -- Note that the parent_process.sensitivity signals
  -- inherit all structural constraints from the structure
  -- level (the parent_process is an indirect specialization
  -- of the process entity from the structure schema).
  -- Therefore the signals are within the architecture
  -- which contains the process, are visible etc.

  ((SIZEOF(outcome) = SIZEOF(parent_wire_wait.sensitivity_signals)) AND
  (QUERY(ws <* outcome |
          ws.parent_wire IN parent_wire_wait.sensitivity_signals)
   = outcome)) OR
   (outcome = []);

  -- The signals which can satisfy the signal_wait are the
  -- sensitivity_signals of the signal_wait.

  concurrent_outcomes:
  QUERY(ws <* outcome | ws.sim_cycle :=: outcome_cycle)
  = outcome;

  -- The wire modifications which satisfied the wait statement
  -- are concurrent.

  QUERY(w <* outcome | w.signal_event) <> [];
  -- There are events on some of the signal_wait
  -- sensitivity_signals.
END_ENTITY;
```

```
FUNCTION compute_signal_wait_outcome_cycle
         (ws: SET OF wire_state): simulation_cycle;

  -- If the signal_wait is satisfied (ws <> []) then
  -- the simulation_cycle of this event is the sim_cycle
  -- of the states of the wires the signal_wait depends on.
  -- (see the rule nearest_signal_wait_outcome)

  IF ws = [] THEN RETURN(?);
           ELSE RETURN(ws[LOINDEX(ws)].sim_cycle);
  END_IF;
END_FUNCTION;

RULE nearest_signal_wait_outcome FOR
    (signal_wait_transaction, wire_state);

-- Between a signal_wait and its outcome_cycle, if this exists,
-- there is no wire_state such that the parent_wire is in the
-- sensitivity_signals of the signal_wait and there is an event
-- on the wire. Therefore the signal_wait outcome wire states
-- are the true cause of satisfying the signal_wait.

WHERE
QUERY(w <* signal_wait_transaction |
     NOT EXISTS(w.outcome_cycle) OR
     (QUERY(ws <* wire_state |
           NOT(ws.parent_wire IN w.parent_wire_wait.sensitivity_signals) OR
           NOT(is_after(ws.sim_cycle\Event,
                        w\wait_transaction.sim_cycle\Event) AND
               is_before(ws.sim_cycle\Event,w.outcome_cycle\Event) AND
               ws.signal_event))
     = wire_state))
= signal_wait_transaction;
END_RULE;

ENTITY condition_wait_transaction
       SUBTYPE OF (wait_transaction);

-- Models a condition wait statement, which suspends the
-- parent process, waiting for a condition (depending on
-- some signals visible from the parent process) to become
-- VHDL_true.

  parent_condition_wait: condition_wait;

  outcome: OPTIONAL condition_wait_value;
  -- The value of the condition_wait which is VHDL_true and
  -- satisfies the condition_wait_transaction.

DERIVE
  SELF\wait_transaction.outcome_cycle: simulation_cycle:=
  compute_condition_wait_outcome_cycle(outcome);

WHERE
```

```
  check_condition_wait_outcome(SELF);
  -- The condition_value which satisfies the condition_wait
  -- must be VHDL_true and must belong to the condition_wait.

  parent_condition_wait.parent_process :=:
  SELF\transaction.parent_real_process;

  -- Note that the parent_process.condition signals
  -- inherit all structural constraints from the structure
  -- level (the parent_process is an indirect specialization
  -- of the process entity from the structure schema).
  -- Therefore the signals are within the architecture
  -- which contains the process, are visible etc.
END_ENTITY;

FUNCTION compute_condition_wait_outcome_cycle
         (cv: condition_wait_value): simulation_cycle;

  -- If the condition_wait is satisfied (cv <> ?) then
  -- the simulation_cycle of this event is the sim_cycle
  -- of the condition_wait_value.

  IF NOT EXISTS(cv) THEN RETURN(?);
             ELSE RETURN(cv.sim_cycle);
  END_IF;
END_FUNCTION;

FUNCTION check_condition_wait_outcome
         (cw: condition_wait_transaction): LOGICAL;

  -- The condition_value which satisfies the condition_wait
  -- cw must be VHDL_true and must belong to cw value history.

  IF NOT EXISTS(cw.outcome) THEN RETURN(TRUE); END_IF;
  RETURN((cw.outcome.parent_condition_wait :=:
          cw.parent_condition_wait) AND cw.outcome.effective_value);
END_FUNCTION;

ENTITY condition_wait_value;
-- This models a value taken in a simulation_cycle by a
-- condition_wait statement (in fact the statement predicate).
-- For each condition_wait and for each simulation_cycle there
-- is such a value.

  current_value: VHDL_value;
  -- The value of the condition wait predicate

  condition_wire_states:  SET [1 : ?] OF wire_state;
  -- The wire states on which the parent_condition_wait depends.
  -- Therefore, current_value = f(condition_wire_states).

DERIVE
  parent_condition_wait: condition_wait:=
  parent_condition_wait_history.parent_condition_wait;
```

```
  effective_value:
  LOGICAL:= (current_value = VHDL_true) AND
            (QUERY(ws <* condition_wire_states | ws.signal_event) <> []);

  -- The effective_value is true if the current_value is VHDL_true
  -- and there are some events on the condition signals.

INVERSE
  sim_cycle: simulation_cycle FOR condition_wait_values;

  parent_condition_wait_history: condition_wait_history FOR values;
  -- The condition_wait_history groups all the values the condition
  -- wait takes during the simulation process.

WHERE
  is_boolean_type(current_value.object_type);
  -- The current_value is a VHDl boolean.

  (SIZEOF(condition_wire_states) =
   SIZEOF(parent_condition_wait.condition_signals)) AND
  (QUERY(ws <* condition_wire_states |
            ws.parent_wire IN parent_condition_wait.condition_signals)
  = condition_wire_states);

  -- The signals which can satisfy the condition_wait are the
  -- condition_signals of the condition_wait.

  QUERY(ws <* condition_wire_states | ws.sim_cycle :=: sim_cycle)
  = condition_wire_states;

  -- The wire modifications which satisfy the wait statement
  -- are concurrent.
END_ENTITY;

RULE nearest_condition_wait_outcome FOR
     (condition_wait_transaction,condition_wait_value);

-- Between a condition_wait CW and its outcome_cycle, if this exists,
-- there is no VHDL_true condition_wait_value for CW based on events
-- on the condition predicate signals. Therefore, the value which
-- satisfies the condition_wait is the nearest such value.

WHERE
QUERY(w <* condition_wait_transaction |
        NOT EXISTS(w.outcome_cycle) OR
        (QUERY(v <* condition_wait_value |
            (v.parent_condition_wait :<>: w.parent_condition_wait) OR
            NOT(is_after(v.sim_cycle\Event,
                         w\wait_transaction.sim_cycle\Event) AND
                is_before(v.sim_cycle\Event,w.outcome_cycle\Event) AND
                v.effective_value))
         = condition_wait_value))
= condition_wait_transaction;
```

```
END_RULE;


ENTITY delay_wait_transaction
       SUBTYPE OF (wait_transaction);

  parent_delay_wait: delay_wait;
  delay_wait_outcome: delay_wait_transaction_outcome;

DERIVE
  due_time: Time_value:=
  SELF\transaction.sim_cycle.event_time + parent_delay_wait.delay;

  SELF\wait_transaction.outcome_cycle:simulation_cycle:=
  delay_wait_outcome.sim_cycle;

WHERE
  parent_delay_wait.parent_process :=: parent_real_process;

  outcome_cycle\Event.event_time = due_time;
  -- The wait outcome time_value is the projected time of SELF.
END_ENTITY;

ENTITY delay_wait_transaction_outcome;

-- This models the event of satisfying a delay_wait
-- and is used within the simulation_cycle entity for
-- synchronizing purposes.

INVERSE
  sim_cycle: simulation_cycle FOR delay_wait_outcomes;
END_ENTITY;

FUNCTION is_signal_wait_transaction(x: wait_transaction): LOGICAL;
  RETURN(is_type(x,'BEHAVIOUR_SCHEMA.SIGNAL_WAIT_TRANSACTION'));
END_FUNCTION;

FUNCTION is_delay_wait_transaction(x: wait_transaction): LOGICAL;
  RETURN(is_type(x,'BEHAVIOUR_SCHEMA.DELAY_WAIT_TRANSACTION'));
END_FUNCTION;

RULE blocked_process FOR (wait_transaction,transaction);
-- No transaction for a process P can occur while a wait_transaction
-- for P is still in progress (not satisfied). P is blocked.

WHERE
QUERY(wt <* wait_transaction |
      NOT EXISTS(wt.outcome_cycle) OR
      (QUERY(t <* transaction |
            (wt\transaction.parent_real_process :<>: t.parent_real_process) OR
            NOT(is_after(t.sim_cycle\Event,wt\transaction.sim_cycle\Event) AND
                is_before(t.sim_cycle\Event,wt.outcome_cycle)))
      = transaction))
= wait_transaction;
```

```
END_RULE;

ENTITY assign_transaction
       SUBTYPE OF (transaction);

-- This models the result of executing a synthetic signal
-- assignment (a sequence of assignments for a single target
-- signal) contained within:

--   a) a process_statement (subtype of process);
--   b) a concurrent_assignment (subtype of process).

  parent_wire_assignment: wire_assignment;

  -- The attributes
  -- target: wire, and parent_process: real_process
  -- are inherited from wire_assignment which is
  -- indirectly a signal_or_port_source. By force
  -- of this inheritance the target is a controlled
  -- signal of the parent_process.

  wait_cause: OPTIONAL wait_transaction;

  -- The cause of executing a signal assignment is a pending
  -- wait_transaction satisfied in the current simulation cycle
  -- (except for some assignments at the beginning of simulation).
  -- Note that if the signal assignment occurs within a process
  -- equivalent to a concurrent_assignment then only signal_waits
  -- are possible as wait_cause.

  guard_wire_state: OPTIONAL wire_state;
  -- The wire_state of the guard signal, if this exists.

  assign_effects: LIST OF assign_effect;
  -- This is the list of the effects of executing the
  -- particular assignments represented by the synthetic
  -- signal assignment. An assign_effect itself
  -- can inject several values into the waveform of
  -- the parent wire (real signal or port).

  -- Since there can be multiple assignments to the same
  -- wire while a real_process is active LIST instead of SET
  -- is used to model the order in which these assignments are
  -- executed (their textual order within the process).

DERIVE
  guarded_assignment: LOGICAL:=
  parent_wire_assignment.parent_process.is_guarded;

  -- A signal assignment contained within a process_statement
  -- is not guarded !. Only a signal assignment within a
  -- concurrent_statement can be guarded.

  target: wire:= parent_wire_assignment.target;
```

```
  guarded_target: LOGICAL:= is_guarded_signal(target\signal_or_port);
  -- Is the assignment target register or bus ?

INVERSE
  parent_assign_driver: assign_driver FOR current_assign;
  -- An assign_transaction is member of an assign_driver.

WHERE
  SELF\transaction.parent_real_process :=:
  parent_wire_assignment.parent_process;

  -- The transaction parent_process is also the parent
  -- process of the assignment.

  check_guard_ownership_and_time(SELF);

  -- The guard_wire_state must be the state of the
  -- guard_signal wire of the wire_assignment which
  -- generated the transaction. In addition, the
  -- guard_wire_state must be concurrent with the
  -- SELF\transaction.

  (NOT is_simulation_start_cycle(SELF\transaction.sim_cycle) AND
   EXISTS(wait_cause)) OR
  (is_simulation_start_cycle(SELF\transaction.sim_cycle) AND
   NOT EXISTS(wait_cause));

  -- At the start of simulation an assignment (depending on its
  -- position in the process) can be executed straight on.
  -- After the start of simulation, an assignment is executed
  -- when its parent process becomes unblocked as the effect
  -- of satisfying a wait_transaction generated in a previous
  -- simulation_cycle.
END_ENTITY;

TYPE transport_or_inertial = ENUMERATION OF (transport,inertial);
END_TYPE;

ENTITY projected_value;
  new_value: VHDL_value;
  projected_time: Time_value;
END_ENTITY;

ENTITY assign_effect;
  -- This represents the effect of executing a
  -- single signal assignment within a process.

  assign_mode: transport_or_inertial;
  assign_values: LIST OF projected_value;

INVERSE
  parent_assign_transaction: assign_transaction FOR assign_effects;
```

```
WHERE
  time_ordered_values:
  check_assigned_values_time_ordering(assign_values);
  -- The list of projected values must be ordered
  -- according to the projected_time.

  valid_scheduled_values:
  QUERY(pv <* assign_values | pv.projected_time >=
                              parent_assign_transaction\
                              transaction.sim_cycle\
                              Event.event_time)
  = assign_values;
  -- The assign_values must be part of the "future" of the
  -- assign_transaction which created them. They do not
  -- occur in the current simulation cycle, even if the
  -- their projected_time is equal to the current sim_cycle
  -- time (0 delays) !

  consistent_type:
  QUERY(pv <* assign_values |
            pv.new_value.object_type =
            parent_assign_transaction.parent_wire_assignment.object_type)
  = assign_values;

  -- The value assigned to a wire (and implicitly the
  -- waveform_values of that wire) have the type of the
  -- signal or port the wire corresponds to. The constraint
  -- between the parent_assignment type and the
  -- parrent_assignment.target signal was specified at the
  -- structure level (see signal_or_port entity and
  -- signal_or_port_sources which are important inheritance
  -- roots for the wire, etc.).
END_ENTITY;

FUNCTION check_assigned_values_time_ordering
         (pv: LIST OF projected_value): LOGICAL;

-- Checks the ascending projected_time ordering
-- of the pv set.

  LOCAL i: INTEGER; END_LOCAL;
  REPEAT i:= LOINDEX(pv)+1 TO HIINDEX(pv) BY 1;
    IF pv[i].projected_time <= pv[i-1].projected_time
    THEN RETURN(FALSE);
    END_IF;
  END_REPEAT;
  RETURN(TRUE);
END_FUNCTION;

FUNCTION check_guard_ownership_and_time
        (t: assign_transaction): LOGICAL;

  IF t.parent_wire_assignment.parent_process.is_guarded
  THEN RETURN((t.guard_wire_state.parent_wire :=:
```

```
               t\transaction.parent_real_process.guard_signal) AND
              (t.guard_wire_state.sim_cycle :=:
               t\transaction.sim_cycle));
  ELSE RETURN(TRUE);
  END_IF;
END_FUNCTION;

-- END_SCHEMA;
```

9.9 Simulation schema

```
-- SCHEMA simulation_schema;
-- =======================================================
-- This schema is the counterpart of structure_schema but
-- this time from the 'time' point of view.
-- =======================================================

ENTITY wire_history;
  wire_states: SET OF wire_state;

INVERSE
  parent_wire: wire FOR behaviour;
  parent_simulation: simulation_process FOR wires_history;
END_ENTITY;

ENTITY condition_wait_history;
  values: SET OF condition_wait_value;

INVERSE
  parent_condition_wait: condition_wait FOR behaviour;
END_ENTITY;

ENTITY real_process_history;
  transactions: SET OF transaction;

INVERSE
  parent_real_process: real_process FOR behaviour;
  parent_simulation: simulation_process FOR processes_history;
END_ENTITY;

ENTITY simulation_cycle SUBTYPE OF(Event);
  wire_states: SET OF wire_state;
  transactions: SET OF transaction;
  condition_wait_values: SET OF condition_wait_value;
  delay_wait_outcomes: SET OF delay_wait_transaction_outcome;

DERIVE
  assign_drivers:
  SET OF assign_driver:= cycle_assign_drivers(wire_states);

  epv: projected_value:= earliest_projected_value(assign_drivers);
```

```
INVERSE
  parent_simulation: simulation_process FOR simulation_cycles;

  -- Since there is an unique simulation_process
  -- all simulation cycles belong to that process.

WHERE
  no_initial_transactions:
  (is_simulation_start_cycle(SELF) AND (transactions = [])) OR
  (NOT is_simulation_start_cycle(SELF));

  -- There are no transactions at the beginning of
  -- the simulation process.

  cycle_scheduling:
  (delay_wait_outcomes <> []) OR
  ((epv <> ?) AND (SELF\Event.event_time = epv.projected_time)) OR
  is_simulation_end_cycle(SELF);

  -- If the cycle contains delay wait outcomes then its time
  -- value is computed according to the delay transactions
  -- satisfied by the outcomes. (See the delay_wait_transaction).

  -- If there are projected values in the drivers of this
  -- simulation cycle and the cycle does not contain delay wait
  -- outcomes then the simulation cycle time value is the
  -- earliest projected value.

  -- If there are no projected values (i.e. all waveforms are
  -- empty) and no pending delay waits then the cycle must be
  -- the last cycle of the simulation process.
END_ENTITY;

FUNCTION is_simulation_end_cycle(cycle: simulation_cycle): LOGICAL;
  RETURN(cycle :=: cycle.parent_simulation.simulation_end_cycle);
END_FUNCTION;

FUNCTION is_simulation_start_cycle(cycle: simulation_cycle): LOGICAL;
  RETURN(cycle :=: cycle.parent_simulation.simulation_start_cycle);
END_FUNCTION;

-- Note that (NOT is_simulation_start_cycle(c)) implies that
-- is_after(c\Event,c.parent_simulation.simulation_start_cycle\Event)
-- This property is forced by the simulation_flow constraint from
-- the simulation_process entity.

ENTITY simulation_process;
  simulated_design: architecture_design;
  simulation_cycles: SET OF simulation_cycle;

  simulation_start_cycle: simulation_cycle;
  simulation_end_cycle: simulation_cycle;
  simulation_time_limit: OPTIONAL Time_value;
```

```
  wires_history: SET OF wire_history;
  processes_history: SET OF real_process_history;
  condition_waits_history: SET OF condition_wait_history;

DERIVE
  wires: SET OF wire:= simulated_design.wires;
  processes: SET OF real_process:= simulated_design.real_processes;

  simulation_start_time:
  Time_value:= simulation_start_cycle\Event.event_time;

WHERE
  start_of_simulation:
  (simulation_start_cycle IN simulation_cycles) AND
  (simulation_start_cycle\Event.event_time >= start_time) AND
  (QUERY(cycle <* simulation_cycles |
             (cycle :=: simulation_start_cycle) OR
             is_after(cycle,simulation_start_cycle))
   = simulation_cycles);

  -- The simulation_start_cycle is unique in the set of
  -- simulation_cycles. Note that (simulation_start_cycle IN
  -- simulation_cycles) is always true because of the unicity
  -- of the simulation_process. It is here only for clarity.

  end_of_simulation:
  (simulation_end_cycle IN simulation_cycles) AND
  dormant_cycle(simulation_end_cycle,simulation_time_limit) AND
  (QUERY(cycle <* simulation_cycles |
             (cycle :=: simulation_end_cycle) OR
             (is_before(cycle,simulation_end_cycle) AND
              NOT dormant_cycle(cycle,simulation_time_limit)))
   = simulation_cycles);

  -- The end_cycle is the first simulation cycle which is dormant
  -- while all its predecessors are not dormant.
  -- Note that (simulation_end_cycle IN simulation_cycles) is always
  -- true because of the unicity of the simulation_process.
  -- It is here only for clarity.

  simulation_flow:
  QUERY(cycle1 <* simulation_cycles |
        (cycle1 :=: simulation_start_cycle) OR
        (SIZEOF(QUERY(cycle2 <* simulation_cycles |
                      is_next(cycle1\Event,cycle2\Event))) = 1))
  = simulation_cycles;

  -- Except for the simulation_start_cycle, each simulation cycle
  -- has exactly one direct predecessor in the set of simulation_cycles.

  valid_simulation_time_limit: check_simulation_time_limit(SELF);
  -- If specified, the simulation_time_limit must
  -- be equal to or greater than the simulation_time_start.
```

```
  (SIZEOF(wires) = SIZEOF(wires_history)) AND
  (QUERY(wh <* wires_history |
         SIZEOF(QUERY(w <* wires | w :=: wh.parent_wire)) = 1)
   = wires_history);

  -- Each wire of the simulated_design must have a
  -- wire_history in the simulation_porcess.

  (SIZEOF(processes) = SIZEOF(processes_history)) AND
  (QUERY(ph <* processes_history |
         SIZEOF(QUERY(p <* processes | p :=: ph.parent_process)) = 1)
   = processes_history);

  -- Each process of the simulated_design must have a
  -- process_history in the simulation_process.


  QUERY(cycle <* simulation_cycles |
        QUERY(wh <* wires_history |
                 SIZEOF(QUERY(ws <* wh.wire_states |
                                  ws.sim_cycle :=: cycle)) = 1)
        = wires_history)
  = simulation_cycles;

  -- For each simulation_cycle C and for each wire_history WH
  -- there is a wire_state WS IN WH such that WH occurs at C.
  -- Since the wire states in WH belong to the same wire, it
  -- follows that the wire is present in each simulation_cycle
  -- of the simulation_process.

  QUERY(cycle <* simulation_cycles |
        QUERY(ch <* condition_waits_history |
                 SIZEOF(QUERY(v <* ch.values |
                                 v.sim_cycle :=: cycle)) = 1)
        = condition_waits_history)
  = simulation_cycles;

  -- For each simulation_cycle C and for each condition_wait_history CH
  -- there is a condition_wait_value V IN CH such that V occurs at C.
  -- Therefore a condition_wait has a value in each simulation cycle.
END_ENTITY;

RULE unique_simulation_process FOR (simulation_process);
WHERE
  SIZEOF(simulation_process) = 1;
END_RULE;

FUNCTION check_simulation_time_limit
         (sim_process: simulation_process): LOGICAL;

  IF NOT EXISTS(sim_process.simulation_time_limit)
  THEN RETURN(TRUE);
  ELSE RETURN(sim_process.simulation_time_limit >=
              sim_process.simulation_start_time);
```

```
  END_IF;
END_FUNCTION;

FUNCTION dormant_cycle
         (cycle: simulation_cycle;
          simulation_time_limit: Time_value): LOGICAL;

-- A cycle is 'dormant' if its time is >= simulation_end_time
-- or if the cycle has only void assign_drivers, no transactions
-- and no delay_wait_outcomes.

  LOCAL active_assign_drivers: SET OF assign_driver:=
        QUERY(ad <* cycle.assign_drivers |
                  ad.current_waveform.waveform_values <> []);
  -- Assign drivers which contain non empty waveforms
  END_LOCAL;

  IF EXISTS(simulation_time_limit)
  THEN
      IF cycle\Event.event_time > simulation_time_limit
      THEN RETURN(TRUE);
      END_IF;
  END_IF;

  RETURN((active_assign_drivers = []) AND
         (cycle.transactions = []) AND
         (cycle.delay_wait_outcomes = []));
END_FUNCTION;

FUNCTION cycle_assign_drivers
         (ws: SET OF wire_state): SET OF assign_driver;
-- Returns all the assign_drivers contained within the wire
-- states in a simulation cycle.

  LOCAL i,j: INTEGER;
        ad: SET OF driver:= [];
        d: SET OF driver;
  END_LOCAL;

  REPEAT i:= LOINDEX(ws) TO HIINDEX(ws) BY 1;
    d:= ws[i].drivers;
    REPEAT j:= LOINDEX(d) TO HIINDEX(d) BY 1;
      IF is_assign_driver(d[j])
      THEN ad:= ad + [d[j]];
      END_IF;
    END_REPEAT;
  END_REPEAT;
  RETURN(conv_type(ad));
END_FUNCTION;

FUNCTION earliest_projected_value
         (ad: SET OF assign_driver): projected_value;
-- Returns the earliest projected_value out of
-- all projected values in the ad set of assign drivers.
```

```
  LOCAL i: INTEGER;
        pv: projected_value;
        epv: projected_value:= ?;
  END_LOCAL;

  REPEAT i:= LOINDEX(ad) TO HIINDEX(ad) BY 1;
    pv:= ad[i].current_waveform.earliest_value;
    IF pv <> ?
    THEN
         IF epv = ?
         THEN epv := pv;
         ELSE
              IF epv.projected_time > pv.projected_time
              THEN epv:= pv;
              END_IF;
         END_IF;
    END_IF;
  END_REPEAT;

  RETURN(epv);
END_FUNCTION;

-- END_SCHEMA;
```

9.10 Time schema

```
-- SCHEMA time_schema;
-- ======================================================
-- This is already done. It is the linear-measurable-
-- instantaneous-event time-model.
-- ======================================================

-- start_time CONSTANT declaration is here
-- simulation_start CONSTANT declaration is here

TYPE Time_value = REAL;
END_TYPE;

ENTITY Event;
  event_time_point: Time_point;

DERIVE
  event_time: Time_value := event_time_point.time;
END_ENTITY;

ENTITY Time_point;
  present: SET [1: ?] OF Event;
  past: SET OF Event;
  time: Time_value;

WHERE
```

```
  valid_time_value:
  is_equal(time,start_time) OR
  ((past <> []) AND is_greater(time,start_time));

  event_ordering:
  SIZEOF(QUERY (e <* past |
                is_less(e.event_time,time) OR
                is_equal(e.event_time,time))) = SIZEOF(past);

  -- events with the same event_time can be contained in both
  -- the present and the past of the time point
END_ENTITY;

RULE unique_present FOR (Time_point, Event);
WHERE
QUERY(t <* Time_point |
        QUERY(e <* Event | e.event_time_point :=: t) = t.present)
= Time_point;
END_RULE;

RULE linearise_time FOR (Time_point);
--  for any t, t1 IN Time_point, t <> t1,
--      either t must be after t or t1 must be after t
WHERE
QUERY(t <* Time_point |
         QUERY (t1 <* Time_point |
                     (t :=: t1) OR
                     (t.past + t.present <= t1.past) OR
                     (t1.past + t1.present <= t.past))
         = Time_point)
= Time_point;
END_RULE;

FUNCTION is_next(e1,e2: Event): LOGICAL;
RETURN (e1.event_time_point.past =
       (e2.event_time_point.present + e2.event_time_point.past));
END_FUNCTION;

FUNCTION is_concurrent(e1,e2: Event): LOGICAL;
RETURN (e1.event_time_point.present = e2.event_time_point.present);
END_FUNCTION;

FUNCTION is_previous(e1,e2: Event): LOGICAL;
RETURN (e2.event_time_point.past =
       (e1.event_time_point.present + e1.event_time_point.past));
END_FUNCTION;

FUNCTION is_before(e1,e2: Event): LOGICAL;
RETURN ((e1.event_time_point.present + e1.event_time_point.past) <=
         e2.event_time_point.past);
END_FUNCTION;

FUNCTION is_after(e1,e2: Event): LOGICAL;
RETURN ((e2.event_time_point.present + e2.event_time_point.past) <=
```

```
         e1.event_time_point.past);
END_FUNCTION;

FUNCTION is_greater(ta,tb: Time_value):BOOLEAN;
RETURN (ta > tb);
END_FUNCTION;

FUNCTION is_less(ta,tb: Time_value):BOOLEAN;
RETURN (ta < tb);
END_FUNCTION;

FUNCTION is_equal(ta,tb: Time_value):BOOLEAN;
RETURN (ta = tb);
END_FUNCTION;

END_SCHEMA;
```

Acknowledgements

Support for the work described in this report has been received from the Commission of the European Communities through the ECIP (ESPRIT 2072) project. The help of colleagues in the CAD group at the University of Manchester is greatly appreciated. I wish to thank in particular H.J. Kahn (Manchester University) and Wolfgang Mueller (Paderborn University) for their useful comments to the report.

References

[IE88] IEEE. IEEE Standard VHDL Language. Reference Manual, IEEE, Inc., 345 East 47th Street, New York, NY 10017, USA, 1988

[ISO91] ISO TC184/SC4/WG5 EXPRESS Language Reference Manual. ISO10303:Part 11 Version N14, April 1991

[LSU89] Lipsett R., Schaefer C. and Ussery C. VHDL: Hardware Description and Design. Kluwer Academic Publishers, 1989

[GC92-1] Giumale C. A. Models of Time. Report ECIP2/UM/009-1, University of Manchester, 1992

[GCKH93] Giumale C. A. and Kahn H.J. An Information Model of Time. DAC Conference, Dallas 14-18 June 1993

[GC92-2] Giumale C. A. Towards a VHDL Model. Report ECIP2/UM/008-1, University of Manchester, 1992

[MR92] Marshall R.A.J. and Kahn J.H., A Structural Information Model of VHDL. In Jean Mermet, editor, *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*. Kluwer academic publishers, 1992.

[SO93] Olcoz S. and Colom J.M. VHDL Through the Looking Glass VHDL-Formum, Innsbruck March 1993

[SMW92] Schaefers L., Mueller W. and Wilkes W. Examination of Concepts in Existing Modelling Languages / Methodologies to Model Behavioural Semantics. Deliverable Report ECIP2/HU/008-1, 1992

[VT92] Van Tassel J. P. A Formalistation Of The VHDL Simulation Cycle. Technical Report No.249, Computer Laboratory, University of Cambridge, 1992

[WD92] Wilsey P. A. and Davis K. A. Modeling VHDL's Execution Semantics in EXPRESS. Unpublished paper, 1992