5 Conclusions

The presented model had two objectives. First to devise and use a model of time - based on events - model which attaches specific semantics to the time_units metric of the time space. In absence of such a model, integers or whatsoever time_units would stand just for values common to a large class of objects. As for specifying data types, a set of values can be seen as the base type of a vector, list, bag, stack, queue etc. Only by constraining the use of these elements one can claim to have specified a particular kind of data type object.

Second, the model emphasised the fact that using appropriate model concept¡s (for example, transactions and states) the behaviour modelling becomes feasible with no extensions added to EXPRESS. It shows that EXPRESS is not a 'static' modelling language because time can be modelled in the same way as any geometrical space ordering concept.

6 Acknowledgements

Support for the work described in this report has been received from the Commission of the European Community 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 Alan R. Williams, H.J. Kahn, Gerhard Scholz (Fernuniversitat Hagen) and Wolfgang Mueller (Paderborn University) for their useful comments and suggestions 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

[KS86] Kowalski R. and Sergot M. A Logic-Based Calculus for Events. New Generation Computing, Vol. 4, 1986, pp 67-95

[LSU89] Lipsett R., Schaefer C. and Ussery C. VHDL: Hardware Description and Design. Kluwer Academic Publishers, 1989

[GA78] Gale R.M. (ed.) The Philosophy of Time. Humanities Press, 1978

[GC92] Giumale C. A. Models of Time. Report ECIP2/UM/009-1, University of Manchester, 1992

[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 Unpublished paper, 1992

```
  QUERY(t <* Time_point |
        QUERY(e <* Event | e.event_time_point :=: t) = t.present)
  = Time_point;
END_RULE;

RULE linearise_time FOR (Time_point);
WHERE
  QUERY(t <* Time_point |
        QUERY(tp <* Time_point | (t = tp) OR
                                 ((t.past + t.present) <= tp.past) OR
                                 ((tp.past + tp.present) <= t.past))
        = Time_point)
  = Time_point;
END_RULE;

RULE big_bang FOR (Time_point);
WHERE
  SIZEOF(QUERY(t <* Time_point | (t.time = time_start) AND
                                 (t.past = []))) = 1;
END_RULE;

FUNCTION is_next(e1,e2: Event): BOOLEAN;
  RETURN (e1.event_time_point.past =
          (e2.event_time_point.present + e2.event_time_point.past));
END_FUNCTION;

FUNCTION is_concurrent(e1,e2: Event): BOOLEAN;
  RETURN (e1.event_time_point.present = e2.event_time_point.present);
END_FUNCTION;

FUNCTION is_previous(e1,e2: Event): BOOLEAN;
  RETURN (e2.event_time_point.past =
          (e1.event_time_point.present + e1.event_time_point.past));
END_FUNCTION;

FUNCTION is_before(e1,e2: Event): BOOLEAN;
  RETURN ((e1.event_time_point.present + e1.event_time_point.past) <=
            e2.event_time_point.past);
END_FUNCTION;

FUNCTION is_after(e1,e2: Event): BOOLEAN;
  RETURN ((e2.event_time_point.present + e2.event_time_point.past) <=
            e1.event_time_point.past);
END_FUNCTION;
END_SCHEMA;
```

```
ENTITY microsecond SUBTYPE OF(Time_value);
END_ENTITY;

ENTITY millisecond SUBTYPE OF(Time_value);
END_ENTITY;

ENTITY second SUBTYPE OF(Time_value);
END_ENTITY;

ENTITY minute SUBTYPE OF(Time_value);
END_ENTITY;

ENTITY hour SUBTYPE OF(Time_value);
END_ENTITY;

 -- is_equal, is_less, is_greater, time_diff, time_add
 -- have to be defined to work with Time_values


ENTITY Event;
  event_time_point: Time_point;

DERIVE
  event_time: Time_value := event_time_point.time;
END_ENTITY;

ENTITY Time_point;
  past: SET OF Event;
  present: SET [1: ?] OF Event;
  time: Time_value;

WHERE
  valid_Time_value:
  is_greater(time,time_start) OR is_equal(time,time_start);

  directional_time: present * past = [];

  event_ordering:
  QUERY(e <* past | is_less(e.event_time,time) OR
                    is_equal(e.event_time,time)) = 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
```

```
ENTITY simulation_process;
  simulation: SET OF simulation_slice;
  architecture: device_architecture;

WHERE
  initial_state:
        SIZEOF(QUERY(sl <* simulation | NOT EXISTS(sl.previous))) = 1;

  -- there is only on simulation slice with previous simulation
  -- cycle undefined

  simulation_membership:
        SIZEOF(QUERY(sl <* simulation |
                        sl.architecture :<>: architecture)) = 0;

  simulation_flow:
        QUERY(sl <* simulation |
                SIZEOF(QUERY(slp <* simulation |
                                sl.current :=: slp.previous)) = 1)
        = simulation;
END_ENTITY;
END_SCHEMA;
```

4.4 Time level model

```
SCHEMA Time_Schema;

CONSTANT time_start: time_type:= 0;
END_CONSTANT;

TYPE time_type = INTEGER;
END_TYPE;

ENTITY Time_value ABSTRACT SUPERTYPE OF
      (ONEOF (fentosecond,picosecond,nanosecond,
              microsecond,millisecond,second,minute,hour));
time: time_unit;
END_ENTITY;

ENTITY fentosecond SUBTYPE OF(Time_value);
END_ENTITY;

ENTITY picosecond SUBTYPE OF(Time_value);
END_ENTITY;

ENTITY nanosecond SUBTYPE OF(Time_value);
END_ENTITY;
```

```
  signal_drivers_current_value_constraint: notdefined; -- function
  (*
     for each signal_status s IN current.signals
       for each d IN s.drivers

           d.current_value =

         if is_type_of(d,'signal_driver')
         then s'.effective_value
              where s' IN current.signals
              such that s'.target_signal :=: d.source

         else if s.active_signal &&
                earliest_time(d.waveform) = current.event_time
              then earliest_value(d.waveform)
              else d'.current_value
                   where s' IN prev_cycle.signals && d' IN s'.drivers,
   is_type_of(d','process_driver'),
                   such that s'.target_signal :=: s.target_signal &&
                   d'.source :=: d.source (same driving process)
  *)

  new_wait_transactions_constraint:
          SIZEOF(QUERY(new_wt <* new_waits |
                  SIZEOF(QUERY(wt <* prev_cycle.satisfied_waits |
                                  wt.source :=: new_wt.source)) = 1))
          = SIZEOF(new_waits);
  (*
     - let new_waits be
       current.wait_transactions - prev_cycle.wait_transactions

       then for each t IN new_waits there must exist one
                t' IN prev_cycle.satisfied_waits
                with t'.source :=: t.source
       (a new wait transaction must have a cause which is a previous
       wait satisfied transaction from the same process)
  *)

  old_satisfied_waits_are_discarded:
        SIZEOF(prev_cycle.transactions.satisfied_waits *
               current.transactions.wait_transactions) = 0;

  non_satisfied_waits_are_preserved:
        (prev_cycle.wait_transactions - prev_cycle.satisfied_waits) <=
        current.wait_transactions;
END_ENTITY;
```

```
DERIVE
  prev_cycle: simulation_cycle:= NVL(previous, init_cycle);

  -- prev_cycle:= IF EXISTS(previous) THEN previous ELSE init_cycle
  -- init_cycle is a constant which describes the initial state of
  -- the simulation process.

  new_waits: SET OF wait_transaction:=
        current.wait_transactions - previous.wait_transactions;

WHERE
  cycles_are_ordered:
        is_next(current,prev_cycle);

  cycles_belong_to_the_same_architecture:
        (current.architecture :=: architecture) AND
        (prev_cycle.architecture :=: architecture);

  effective_values_constraint: notdefined; -- function
  (*
      for each signal_status s IN current.signals

      - s.effective_value =
        if s.active_signal &&
           s.earliest_value.time = current.event_time
        then
           if target_signal.resolved
           then s.resolving_transaction.effective_result
           else s.earliest_value.projected_value
        else s'.effective_value
             where s' IN prev_cycle.signals
             such that s'.target_signal :=: s.target_signal
  *)

  signal_drivers_waveform_constraint: notdefined; -- function
  (*
     for each signal_status s IN current.signals
       for each d IN s.drivers
           such that is_type_of(d,'process_driver')
         let s' IN prev_cycle.signals &&
             d' IN s'.drivers, is_type_of(d','process_driver'),
             such that s'.target_signal :=: s.target_signal &&
                       d'.source :=: d.source

      - the tuple (d.waveform, d'.waveform, d'.waveform_extension)
        must satisfy the constraint C_SD from section 2
  *)
```

```
                        wt.cycle.architecture :=: architecture))
      = SIZEOF(wait_transactions);

  --  for any wt IN wait_transactions: is_next(cycle,wt.cycle)
  --  (all wait_transactions from this cycle were generated in the past)

  wait_transactions_constraint:
      SIZEOF(QUERY(wt <* wait_transactions |
              SIZEOF(QUERY(p <* architecture.processes |
                           p :=: wt.source)) = 1))
      = SIZEOF(wait_transactions);

  --  there must exist exactly one source process for each wait
  --  transaction

  assign_transaction_cycle_membership_constraint:
      SIZEOF(QUERY(t <* assign_transactions | t.cycle :=: SELF))
      = SIZEOF(assign_transactions);

  --  all assign_transactions must belong to the same simulation
  --  cycle and due to the architecture constraint from the
  --  process_transaction both the target signals and the driving
  --  processes belong to the simulated architecture

  assign_transactions_constraint:
      SIZEOF(QUERY(at <* assign_transactions |
              SIZEOF(QUERY(swt <* satisfied_waits |
                           swt.source :=: at.source)) = 1))
      = SIZEOF(assign_transactions);

  --  for any at IN assign_transactions:
  --      there exists one swt IN satisfied_waits with
  --           swt.source :=: at.source
  --  (each assign_transaction is the effect of executing the
  --  process associated with a satisfied wait_transaction)

  signal_states_cardinality_constraint:
      SIZEOF(signals) = SIZEOF(architecture.signals);

  --  all signals in the architecture must have signal_states
  --  in the current cycle
END_ENTITY;


ENTITY simulation_slice;
  previous: OPTIONAL simulation_cycle;
  current: simulation_cycle;
  architecture: device_architecture;
```

```
  delay_wait_transactions: SET OF delay_wait_transaction:=
        QUERY(t <* transactions |
                is_type_of(t,'delay_wait_transaction'));

  satisfied_waits: SET OF wait_transaction:=
        compute_satisfied_waits(wait_transactions,cycle);

  --   satisfied_waits:=
  --      swt IN wait_transactions such that
  --          is_type_of(swt,'signal_wait_transaction') &&
  --          (NOT EXISTS(swt.condition) || swt.condition) &&
  --          swt.wait_for_signals <= cycle.modified_signals
  --          ||
  --          is_type_of(swt,'delay_wait_transaction') &&
  --          swt.due_time = cycle.event_time

 event_time: Time_value:=
        min_time(active_signals,delay_wait_transactions);

  -- The cycle starts either when new effective values are forced
  -- on signals, or when delay_wait_transaction are satisfied. It
  -- depends on which from the mentioned events comes first.

INVERSE
  curr_slice: simulation_slice FOR current;
  next_slice: simulation_slice FOR previous;

  -- there are two simulation slices:
  --   one in which SELF plays the role of current
  --   other in which SELF plays the role of previous

WHERE
  simulation_flow_constraint:
        is_next(next_slice.current,SELF);
  -- the current_cycle of the next simulation slice is next
  -- (in time) to SELF (redundant according to the cycles are ordered
  -- constraint from simulation_slice entity)

  non_idle_cycle:
        SIZEOF(active_signals) + SIZEOF(satisfied_waits) > 0;

  -- There must be at least some awaken processes or some signals
  -- to be updated

  wait_transactions_time_constraint:
        SIZEOF(QUERY(wt <* wait_transactions |
                        is_next(SELF,wt.cycle) AND
```

```
  -- where the value_of_signal returns the effective_value the
  -- target_signal had in the previous simulation cycle

INVERSE
  cycle: simulation_cycle FOR signals;

WHERE
  architecture_membership_constraint:
        target_signal IN cycle.architecture.signals;
  -- signal_status must refer a signal from the simulated architecture

  drivers_cardinality_constraint:
        SIZEOF(drivers) = SIZEOF(target_signal.sources);
  -- the number of drivers = the number of driven signal sources

  type_constraint:
        effective_value.value_type = signal_type(target_signal);

  resolving_constraint:
        NOT target_signal.resolved OR EXISTS(signal_resolving);

  -- signal_resolving must exist only if target_signal.resolved is TRUE
END_ENTITY;
END_SCHEMA;
```

4.3 Simulation level model

```
SCHEMA Simulation_Schema;
REFERENCE FROM Structure_Schema;
REFERENCE FROM Behaviour_SCHEMA;
REFERENCE FROM Time_Schema;


ENTITY simulation_cycle SUBTYPE OF (Event);

  architecture: device_architecture;
  assign_transactions: SET OF assign_transaction;
  wait_transactions: SET OF wait_transaction;
  signals: SET OF signal_status;

DERIVE
  active_signals: SET OF signal:=
         QUERY(s <* signals | s.active_signal);

  modified_signals: SET OF signal:=
        currently_modified_signals(signals);
  -- = ss.target_signal | ss IN signals, ss.signal_event
```

```
ENTITY resolving_transaction SUBTYPE OF (process_transaction);
  effective_parameters: SET OF all_signal_driver;
  effective_result: signal_value;

DERIVE
  father: computation_template:=
          owner.target_signal.resolving_function;

INVERSE owner: signal_status FOR signal_resolving;

WHERE
  effective_parameters_constraint:
        effective_parameters = owner.drivers;

  effective_result_constraint:
        effective_result.projected_value.value_type =
        signal_type(owner.target_signal);

  --  The following proposition can be proved:
  --      owner.drivers agree  source and current_value.value_type
  --      with father.parameters
END_ENTITY;


ENTITY signal_status;
  target_signal: signal;
  effective_value: signal_value;
   -- the current signal value is effective_value.projected_value
  signal_resolving: OPTIONAL resolving_transaction;
  drivers: SET OF all_signal_driver;

DERIVE
  active_signal: BOOLEAN:=
        SIZEOF(QUERY(sd <* drivers |
                         is_type_of(sd,'process_driver') AND
                         SIZEOF(sd.waveform) > 0)) > 0;

  earliest_value: value_transaction:= min_Time_value(drivers);

  -- is the earliest projected_value due to occur in one of the signal
  -- waveforms (if driven by a process)

  signal_event: BOOLEAN:=
        value_of_signal(target_signal,
                          cycle.curr_slice.previous.signals) :<>:
        effective_value;
```

```
  -- * waveform_extension =  U as.assigned_values
  --                as IN owner.cycle.transactions.assign_transactions
  --                as.target_signal = owner.target_signal


  waveforms: SET OF value_transaction:=
             waveform + waveform_extension;

WHERE
  architecture_membership_constraint:
        source IN owner.target_signal.architecture.processes;

  waveforms_component_membership:
        SIZEOF(QUERY(vt <* waveforms |
                (vt.father.target_signal :=: owner.target_signal) AND
                (vt.father.source :=: source)))
        = SIZEOF(waveforms);

  --  for any vt IN waveform U waveform_extension:
  --      vt.father.target_signal :=: owner.target_signal &&
  --      vt.father.source :=: source

  correctness_constraint: notdefined;
  -- * a signal waveform must be consistent according to
  --   restriction R_SD from section 2

  waveform_component_type_constraint: notdefined;

  -- * for any vt IN waveform,
  --   if owner.target_signal.resolved
  --   then vt.projected_value.value_type = p.parameter_type,
  --        where p IN owner.target_signal.resolving_function.parameters
  --        such that p.source :=: source
  --   else vt.projected_value.value_type =
  --        signal_type(owner.target_signal)

  -- The following propositions can be proved:
  -- (a) for any vt IN waveform_extension
  --     if owner.target_signal.resolved
  --     then vt.projected_value.value_type = p.parameter_type,
  --          where p IN owner.target_signal.resolving_function.parameters
  --           such that p.source :=: source
  --     else vt.projected_value.value_type =
  --          signal_type(owner.target_signal)

  -- (b) source IN owner.target_signal.process_sources
END_ENTITY;
```

```
WHERE
  type_constraint:
father.assign_type :=: projected_value.value_type;
END_ENTITY;


ENTITY all_signal_driver ABSTRACT SUPERTYPE OF
       (ONEOF (process_driver, signal_driver));

  current_value: signal_value;
        -- obeys the constraints specified in simulation_cycle entity

INVERSE
  owner: signal_status FOR drivers;
END_ENTITY;


ENTITY signal_driver SUBTYPE OF (all_signal_driver);
  -- this entity is used to stand for the effective value
  -- of a signal connected to another signal

  source: signal;  -- there are no waveforms associated

WHERE
  architecture_membership_constraint:
        source IN owner.target_signal.architecture.processes;

  source_membership_constraint:
        source IN owner.target_signal.sources;

  -- The following proposition holds:
  --     if owner.target_signal.resolved
  --     then current_value.value_type = p.parameter_type,
  --          where p IN owner.target_signal.resolving_function.parameters
  --           such that p.source :=: source
  --     else current_value.value_type = signal_type(owner.target_signal)
END_ENTITY;


ENTITY process_driver SUBTYPE OF (all_signal_driver);
  source: process;
  waveform: SET OF value_transaction;
  -- the current cycle waveform

DERIVE
  waveform_extension: SET OF value_transaction:=
assigned_values(owner.target_signal,
                        owner.cycle.assign_transactions);
```

```
TYPE signal_assign_mode = ENUMERATION OF (transport,inertial);
END_TYPE;


ENTITY assign_transaction SUBTYPE OF (process_transaction);
  target_signal: signal;
  assign_type: signal_value;
         -- the type of values assigned to a signal
  assign_mode: signal_assign_mode;
         -- transport, inertial
  sequence_no: INTEGER;
  assigned_values: SET OF value_transaction;

UNIQUE label:sequence_no;

WHERE
  positive_sequence_no: sequence_no > 0;

  target_signal_membership_constraint:
        target_signal IN source.controlled_signals;

  transaction_source_constraint:
        source IN target_signal.process_sources;

  correctness_constraint: notdefined;
  -- * correctness constraint for assigned_values
  -- (see the restriction R_VT from section 2)

  type_constraint: notdefined;
  -- * if target_signal.resolved
  --   then assign_type = p.parameter_type
  --        where p is IN target_signal.resolving_function.parameters
  --        such that p.source = source
  --   else assign_type = signal_type(target_signal)
END_ENTITY;


ENTITY value_transaction;
  projected_value: signal_value;
  delay: Time_value;

DERIVE
  due_time: Time_value:= time_add(father.cycle.event_time,delay);

INVERSE
  father: assign_transaction FOR assigned_values;
```

```
ENTITY process_transaction ABSTRACT SUPERTYPE OF
      (ONEOF (wait_transaction,
              assign_transaction,
              resolving_transaction));

  source: process;
  cycle: simulation_cycle;

WHERE
  source_membership: source IN cycle.architecture.processes;
  -- the transaction must be generated by a process from the
  -- simulated architecture
END_ENTITY;


ENTITY wait_transaction SUPERTYPE OF
      (ONEOF (delay_wait_transaction,
      signal_wait_transaction))
       SUBTYPE OF (process_transaction);
END_ENTITY;


ENTITY signal_wait_transaction SUBTYPE OF (wait_transaction);
  wait_for_signals: SET OF signal;
  condition: OPTIONAL BOOLEAN;

WHERE
  there_is_something_to_wait_for:
        EXISTS(condition) OR SIZEOF(wait_for_signals) > 0;

  membership_constraint:
        wait_for_signals <= source.sensitivity;

  -- therefore, implicitly,
  -- for each s IN wait_for_signals s.architecture :=:
  -- source.architecture
END_ENTITY;


ENTITY delay_wait_transaction SUBTYPE OF (wait_transaction);
  delay: Time_value;

DERIVE
  due_time: Time_value:= time_add(cycle.event_time,delay);
END_ENTITY;
```

```
ENTITY port ABSTRACT SUPERTYPE OF (ONEOF(local_port,actual_port))
            SUBTYPE OF (signal, signal_source, port_template);

  is_source, is_acceptor: BOOLEAN;
END_ENTITY;


ENTITY local_port SUBTYPE OF (port);

DERIVE
  is_source: BOOLEAN:= direction IN [out,inout,buffer];
  is_acceptor: BOOLEAN:= direction = input;
END_ENTITY;


ENTITY actual_port SUBTYPE OF (port);
DERIVE
  is_source: BOOLEAN:= direction = input;
  is_acceptor: BOOLEAN:= direction IN [out,inout,buffer];
END_ENTITY;


ENTITY local_signal SUBTYPE OF (signal);
WHERE
  local_signal_sources_constraint:
        SIZEOF(QUERY(s <* signal_sources | is_type_of(s,'port')))
        = SIZEOF(signal_sources);

  -- * signal_sources of a local_signal can be only ports
END_ENTITY;
END_SCHEMA;
```

4.2 Behaviour level model

```
SCHEMA Behaviour_Schema;
REFERENCE FROM Type_Schema;
REFERENCE FROM Structure_Schema;
REFERENCE FROM Simulation_Schema;
REFERENCE FROM Time_Schema;


ENTITY signal_value;
  value_type: data_type;
  --  current_value: data_type_value;
  -- it is not at all important since we are only modelling
END_ENTITY;
```

```
ENTITY signal ABSTRACT SUPERTYPE OF (ONEOF (local_signal, port))
              SUBTYPE OF (data_type);

  resolving_function: OPTIONAL computation_template;
  architecture: device_architecture;

DERIVE
  process_sources: SET OF process:=
        QUERY(p <* architecture.processes | SELF IN p.controlled_signals);

  signal_sources: SET OF signal:=
        QUERY(s <* architecture.signals |
                SIZEOF(QUERY(c <* architecture.connections |
                         (s :=: c.source) AND (c.target :=: SELF))) = 1);

  sources: SET OF signal_source:=
        signal_sources + process_sources;

  resolved: BOOLEAN:=
        (SIZEOF(sources) > 1) OR EXISTS(resolving_function);
  -- a resolving function can exist even no. of sources = 1,
  -- being used to convert the type of the signal value.

WHERE
  resolving_type_constraint: notdefined;
  --  NOT resolved || resolving_function.result_type = signal_type(SELF);

  resolving_sources_constraint: notdefined;
  (*
     NOT resolved ||
     SIZEOF(QUERY(sr <* sources |
              SIZEOF(QUERY(srp <* resolving_function.parameters |
                           srp :=: sr)) > 0))
     = SIZEOF(sources);

     --   resolved =>
     --   for each p IN sources,
     --   there must exist param IN resolving_function.parameters,
     --   such that param.source = source
  *)

  resolving_function_constraint:
        (SIZEOF(sources) = 1) OR EXISTS(resolving_function);

  -- * the sources type constraints are inferred from the constraints in:
  --   connection and process entities
END_ENTITY;
```

```
                    is_type_of(s,'port') && s.is_source))
     = SIZEOF(sensitivity_signals);

     --  sensitivity can be only local_signals, and source ports
  *)

  consulted_signals_constraint: notdefined;
  (*
    SIZEOF(QUERY(s <* consulted_signals |
                    is_type_of(s,'local_signal') ||
                    is_type_of(s,'port') && s.is_source))
    = SIZEOF(consulted_signals);

    -- consulted_signals can be only
    -- local_signals and source_ports
  *)

  signals_membership_constraint:
        all_signals <= architecture.signals;

  --    all signals mentioned must belong to the
  --   architecture which contains the process
END_ENTITY;


ENTITY parameter; -- resolving function parameter template
  source: signal;
  parameter_type: data_type;
END_ENTITY;


ENTITY computation_template; -- resolving function template
  parameters: SET OF parameter;
  result_type: data_type;

INVERSE
  owner: signal FOR resolving_function;

WHERE
  cardinality_constraint:
        SIZEOF(parameters) = SIZEOF(owner.signal_sources) +
                             SIZEOF(owner.process_sources);
END_ENTITY;


ENTITY signal_source ABSTRACT SUPERTYPE OF (ONEOF (port, process));
END_ENTITY;
```

```
  target_sources_constraint:
        source IN target.sources;

  type_constraint: notdefined;
  (*
     target.resolved &&
     (signal_type(source) = signal_type(p.signal),
         p IN target.resolving_function.parameters
              with p.source :=: source)
     || NOT target.resolved && signal_type(source) =
        signal_type(target);

  -- source type must agree with the target type
  *)
END_ENTITY;


ENTITY process;
  controlled_signals: SET OF signal;
  sensitivity_signals: SET OF signal;
  consulted_signals: SET OF signal;

DERIVE
  all_signals: SET OF signal:=
        controlled_signals +
        sensitivity_signals +
        consulted_signals;

INVERSE
  architecture: device_architecture FOR processes;

WHERE
  controlled_signals_constraint: notdefined;
  (*
     SIZEOF(QUERY(s <* controlled_signals |
                    is_type_of(s,'local_signal') ||
                    is_type_of(s,'port') && s.is_acceptor))
     = SIZEOF(controlled_signals);

     -- controlled_signals can be only local_signals and
     -- acceptor ports
  *)

  sensitivity_signals_constraint: notdefined;
  (*
     SIZEOF(QUERY(s <* sensitivity_signals |
                    is_type_of(s,'local_signal') ||
```

```
INVERSE
  owner_device: device FOR architectures;

WHERE
  signal_architecture_constraint:
        SIZEOF(QUERY(s <* signals | s.architecture :=: SELF))
        = SIZEOF(signals);

  actual_ports_constraint: notdefined;
  --   actual_port[i] and owner_device.ports[i] must agree as type
  --   and direction and SIZEOF(actual_ports) = SIZEOF(owner_device.ports)

  ports_constraint:
        SIZEOF(QUERY(s <* ports |
                 SIZEOF(QUERY(c <* connections |
                           (c.source :=: s) OR (c.target :=:s))) > 0))
        = SIZEOF(ports);

  -- * all ports must be connected
END_ENTITY;


ENTITY connection;
  source,target: signal;

INVERSE
  architecture: device_architecture FOR connections;

WHERE
  self_connection_constraint: source :<>: target;

  source_constraint: notdefined;
  (*
     is_type_of(source,'port') && source.is_source ||
     is_type_of(source,'local_signal');
  *)

  target_constraint:
       (is_type_of(target,'port') AND target.is_acceptor) OR
        is_type_of(target,'local_signal');

  local_local_signal_constraint:
        NOT (is_type_of(source,'local_signal') AND
             is_type_of(target,'local_signal'));

  membership_constraint:
        (source IN architecture.signals) AND
        (target IN architecture.signals);
```

```
ENTITY device;
  library: design_library;
  ports: SET OF port_template;
  architectures: SET OF device_architecture;

WHERE
  -- * library membership constraints
END_ENTITY;


ENTITY component;
  instance: device;
  behaviour: device_architecture;
  library: design_library;
  ports: SET OF local_port;

INVERSE
  architecture: device_architecture FOR components;

WHERE
  behaviour_constraint:
        behaviour IN instance.architectures;

  ports_constraint: notdefined;
  --   ports[i] and instance.ports[i] must agree as type and
  --   direction and SIZEOF(ports) = SIZEOF(instance.ports)

  -- * for each local_port of the component there is a
  --   matching actual_port in instance.actual_ports

  -- * library membership constraints
END_ENTITY;


ENTITY device_architecture;
  local_signals: SET OF local_signal;
  actual_ports: SET OF actual_port;
  processes: SET OF process;
  components: SET OF component;
  connections: SET OF connection;

DERIVE
  local_ports: SET OF local_port:= compute_local_ports(components);
  ports: SET OF port:= compute_ports(local_ports,actual_ports);
  signals: SET OF signal:= compute_signals(local_signals,ports);
```

4 The VHDL outline model

The following is a preliminary model of VHDL at the levels of structure, behaviour, simulation and time. The model is not meant to be complete. It rather outlines in sufficient detail ideas of the way VHDL behaviour can be modelled and how such a model can be interrelated with a structural model of VHDL.

Moreover, the presented model do 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.

Note that:

- in the model comments `&&` and `||` stand for the wise evaluating AND, respectively OR operators;
- the constraints which are "notdefined" will all be coded as functions.

4.1 Structure level model

```
SCHEMA Type_Schema;

-- Data type valid conversions are stated at this level

ENTITY data_type;
END_ENTITY;
END_SCHEMA;


SCHEMA Structure_Schema;
REFERENCE FROM Type_Schema;


FUNCTION is_type_of(e: GENERIC; type_name: STRING): BOOLEAN;
  RETURN (type_name IN TYPEOF(e));
END_FUNCTION;


ENTITY design_library;
END_ENTITY;


TYPE port_direction = ENUMERATION OF (input, out, inout, buffer);
END_TYPE;


ENTITY port_template SUBTYPE OF (data_type);
  direction: port_direction;
END_ENTITY;
```

is impossible since R_VT(vt) is true by hypothesis and in addition if e and e' differ in their sequence no. the encoding of Rule 1. comes into action.

```
case a3) e,e' IN sd => e,e' => sd'
```

is impossible since R_SD(sd) is true by hypothesis.

b) Assume that there is e IN sd' which does not satisfy R_SD_b), therefore there is e' IN sd' such that R_SD_b(e,e') is false.

```
case b1) e,e' IN sd => e,e' IN sd
```

is impossible since sd satisfies R_SD by hypothesis

```
case b2) e IN sd, e' IN vt => e,e' IN sd'
```

is impossible since by applying recursively the encoding of Rule 1 in function value_constraint:

```
e IN sd and e IN sd' => e.t < e'.t for all e' IN vt
```

and, in addition, is_before(e.father.cycle,e'.father.cycle). Therefore e,e' satisfy R_SD_b.

```
case b3) e IN vt, e' IN sd => e,e' IN sd'
```

is equivalent to case (b2)

```
case b4) e,e' IN vt => e,e' IN sd'
```

is impossible since R_VT(vt) is satisfied and if e,e' have different sequence_no then

```
if e.father.sequence_no < e'.father.sequence_no
then e IN sd' => e.t < e'.t
else e IN sd' => e.t > e'.t
```

therefore the restriction R_SD_b(e,e') is satisfied thus contradicting the hypothesis. Therefore e cannot exist.

```
R_SD is:
        let sd be a signal_waveform for a signal s,
            produced by a single source,

        for any e IN sd

        a) for any e' IN sd, e' <> e => e.t <> e'.t
        b) for any e' IN sd,
           e'.t > e.t => is_after(e'.father.cycle,e.father.cycle) or
                         is_concurrent(e'.father.cycle,e.father.cycle) and
                         e'.father.sequence_no >= e.father.sequence_no
```

Restriction (a) shows that for each time t the signal waveform can hold a unique value.
Restriction (b) shows that sd cannot contain elements which rule out other elements.
In addition to the above constraint there is also necessary the following restriction which shows what is meant by a valid set of value_transactions generated by a signal assignment statement (the set of value_transactions the father of which is a given assign_transaction).

```
R_VT is:
        let vt be a set of value_transactions for a signal s
        for any e,e' IN vt,
            e.father.sequence_no <> e'.father.sequence_no =>
            e.t <> e'.t
```

Proposition 3.1

Assume that:

- `sd` is a signal_waveform such that `R_SD(sd)` is satisfied,
- `vt` is a set of value_transactions which satisfies `R_VT` and
- `sd'` is a signal produced from `sd` and `vt` according to the constraints `C_SD`.

then `R_SD(sd')` is satisfied, that is `sd'` is a correct signal_waveform.

The proof is almost obvious according to the constraint function value_constraint.

a) Assume that there is `e IN sd'` which does not satisfy `R_SD_a`, therefore there is `e' IN sd'` with `e'.t = e.t`.

```
        case a1) e IN sd, e' IN vt => e,e' IN sd' or
                 e IN vt, e' IN sd => e,e' IN sd'
```

is impossible according to the encoding of Rule 1 of Algorithm 3.1 in value_constraint.

```
        case a2) e,e' IN vt => e,e' IN sd'
```

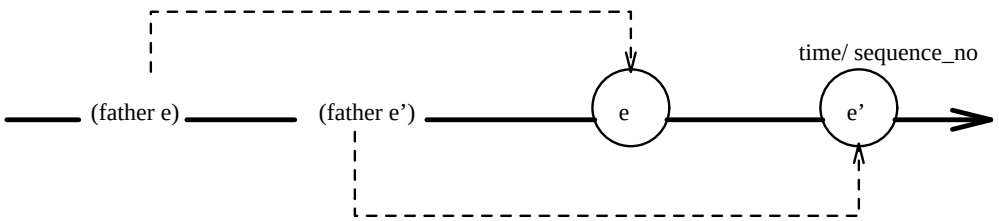

Figure 13: A valid signal waveform

```
   e4       e5
(4,tc), (3,tc+1)
```

and that the s_current_signal_waveform is

```
    e3
(1,tc+3)
```

Apply the constraint `C_SD` for `e IN` {`e1,e2,e3,e4,e5`} according to any order of the set members. For example assume the order for testing `e2,e5,e4,e1,e3`.

```
value_constraint(e2,1,3,e4,e5,e1,e2,e3)
is false since e2 is not valid on level 3 (e3.t < e2.t);

value_constraint(e5,1,3,e4,e5,e1,e2,e3)
is false since e5 is not valid on level 1 (e5.t = e1.t);

e4 does not satisfy the time condition in the current_cycle
(next to prev_cycle), therefore it will be a past value;

value_constraint(e1,1,3,e4,e5,e1,e2,e3)
is false since e1 is not valid on level 2 (e2.value != e1.value);

value_constraint(e3,1,3,e4,e5,e1,e2,e3)
is true since it holds on all levels.
```

The function value_constraint is not a constructor for the data type signal_waveform. It rather encodes an axiom which states the constraints the constructor observes.

However, the constraint `C_SD` is not sufficient for the soundness of the model. It has to be accompanied by a signal_waveform constraint (`R_SD`) which shows what is meant by a correct signal_waveform and also by a restriction (`R_VT`) which shows what is meant by a correct value_transaction.

3.6 Additional restrictions on signal_waveforms and value_transactions

The constraint `C_SD` works properly only if the waveforms are correct. The following restriction specifies the correctness requirements for a waveform (see figure 13).

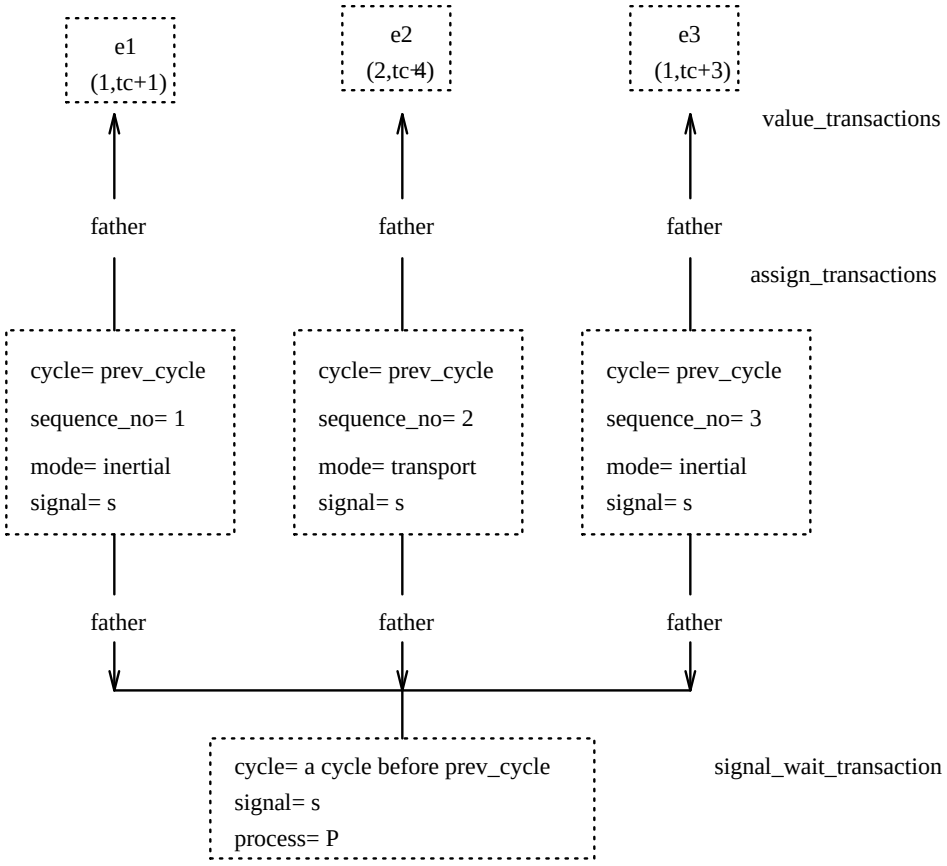

Figure 12: An example of value_constraint application

3.5.2 An example

Consider the following VHDL program fragment:

```
signal s: integer;

P: process
   begin
       s <=  transport 1 after 1;
       s <=  2 after 4;
       s <=  1 after 3;
       wait for s;
   end_process P;
```

Consider that in the prev_cycle (the event_time of which is `tc`) the wait transaction generated by the "wait for s" statement is satisfied. The process `P` resumes execution and generates the set of s_value_transactions shown in figure 12.

Consider that s_prev_signal_waveform is

```
boolean value_constraint
        (e,min_level,level,old_transactions,new_transactions) is
 Let
  level_transactions:= e" | e" IN new_transactions and
                            e".father.sequence_no = level;
  ef.t:=    MIN e'.t;
         e' IN level_transactions

  -- Rule 2 of the Algorithm 1:
  -- the current level value transactions are for sure in the
  -- projected waveform for the current level

  if e IN level_transactions then return true;

  -- Rule 1 of the Algorithm 1:
  -- if e occurs after the earliest value transaction on the current level
  -- then it cannot be in the projected waveform for the current level

  if e.t >= ef.t then return false;
  -- e comes from the past; test if e has not been wiped away in the
  -- past from the current projected transaction

  if level > min_level and
     not value_constraint(e,level-1,old_transactions,new_transactions)
  then return false;

  if ef.father.mode is transport
  then return true;
  else -- ef is an inertial projected value

     if e.value != ef.value
     then return false;
     else
         -- Rules 3 - 6 of Algorithm 1:
         -- for all e' which are valid members of the projected
         -- transaction for the current level of new_transactions and
         -- which, in time, are between e and ef the value component
         -- of e' must be equal to the value component of ef.

         for e' IN old_transactions + new_transactions do
            if e'.father.sequence_no <= level and
               e.t < e'.t and e't < ef.t and
               value_constraint(e',min_level,level,
                                old_transactions,new_transactions)
            then
                if e'.value != ef.value then return false;
     return true;
```

- s_prev_waveform_extension is the set of value transactions for signal s, generated by executing signal assignments from within source p triggered in the previous cycle by a satisfied wait transaction.
- s_current_signal_waveform is the waveform for signal s in the current simulation cycle, it is a set of value_transactions for s the source of which is p.

3.5 Informal specification of the waveform relationship

The relationship will be specified as a constraint which should observe the following algorithm [IE88] for computing a new signal waveform from an old signal waveform according to an atomic value_transaction (ie. a transaction which is the effect of a single signal assignment statement).

```
Algorithm 3.1
```

1. All old transactions (values) that are projected to occur at or after the time of the earliest new transaction is projected to occur are deleted from the projected output waveform.
2. The new transactions are then appended to the projected output waveform in the order of their projected occurrence.

— for inertial delay only —

3. All of the new transactions are marked.
4. An old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction.
5. The transaction which determines the current value of the waveform is marked.
6. All unmarked transactions (all of which are old transactions) are deleted from the projected output waveform.

3.5.1 The waveform constraint (C_SD)

```
Define

  max_level= MAX e'.father.sequence_no
             e' IN s_prev_waveform_extension, and

  min_level= MIN e'.father.sequence_no
             e' IN s_prev_waveform_extension.

Then,
  for each e IN s_prev_signal_waveform + s_prev_waveform_extension
     e IN s_current_signal_waveform only if e satisfies:

     e.t > prev_cycle.event_time and
     value_constraint(e,min_level,max_level,s_prev_signal_waveform,
                      s_prev_waveform_extension).
```

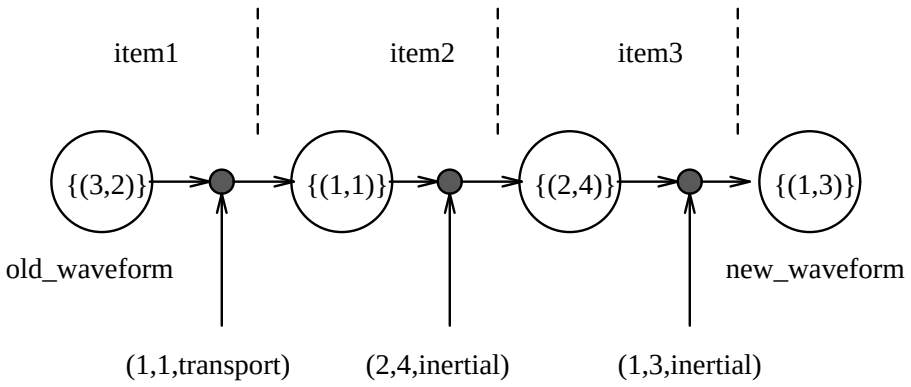

Figure 9: Sequential modelling of signal assignments

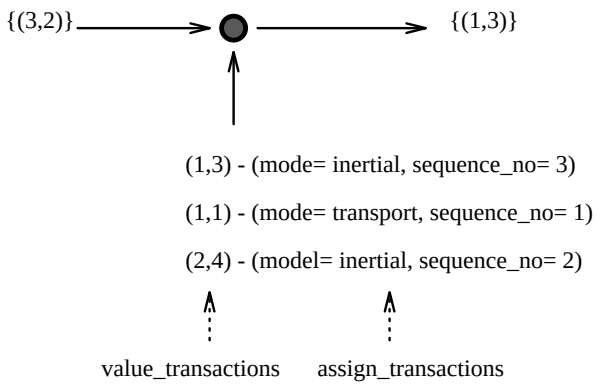

Figure 10: Holistic modelling of signal assignments

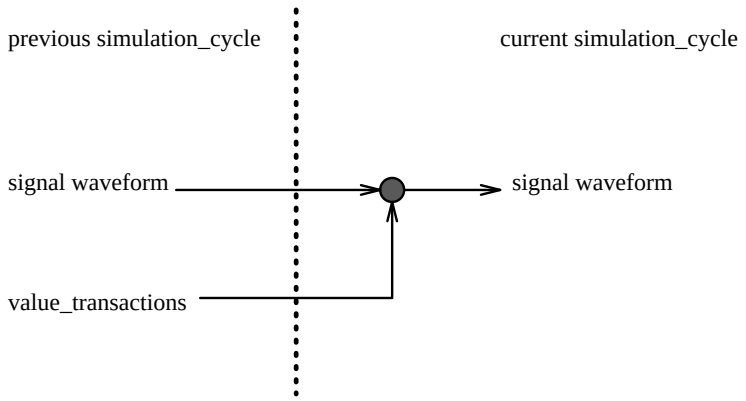

Figure 11: Relationships between two time adjacent signal waveforms

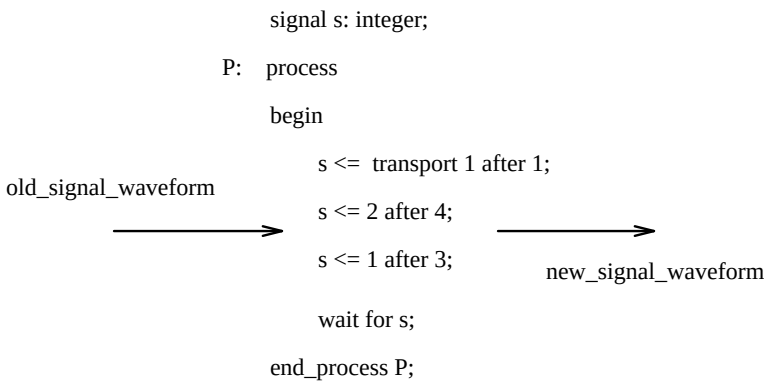

Figure 8: Modification of a signal waveform

- a) to model the sequential transformation of a signal waveform "to the letter" and therefore to model explicitly each atomic modification of the waveform;
- b) to adopt a holistic point of view and hence to model the effect of modifying a signal waveform by all the contributing value_transactions (waveform_ extension) generated in a simulation cycle.

The first approach is inelegant since it introduces artificial data structures (arrays or lists) to capture the sequencing of assign_transactions. Constraints can be asserted between `item[i] - item[i+1]` of the data structure, considering that each item holds the intermediate value of the new _signal_waveform and the atomic value_transaction generated by a single signal assign statement. In addition, there are two particular points for which constraints have to be stated apart from the others: one is at the beginning of the data data structure (constraints between old_signal_ waveform, `item[1].value_transaction` and `item[1].new_signal_waveform`), the other is at the "end" of data structure (constraints between `item[n].new_signal_waveform, item[n].value_transaction` and new_signal_waveform). That will be cumbersome, although the constraints are straightforward. A schemata of this approach for the example in figure 8 is illustrated in figure 9.

The second approach is elegant and more vivid from the modelling point of view. What is important in VHDL from the behavioural point of view is not what happens inside a process, or other way round how a new projected waveform is computed piecewise, but rather the global effect of a process execution on the old signal_waveform. However, the constraints are more elaborate than in case (a).

The second approach of modelling is considered below. As already mentioned, we have to specify the sensible constraints the current signal_waveform has to satisfy when generated according to the previous signal_waveform and previous contributing value_transactions.

`Notations`

- prev_cycle is the previous simulation cycle (it is an Event instance);
- s_prev_waveform is the waveform for signal s in the previous simulation cycle, it is a set of value_transactions for s produced by a single signal source p;

- In addition to the above relationships there is a difficult one. Assume that `d` is a process_driver with the source `p` (a process) and a target_signal `s` in the previous simulation cycle and `d'` be the a process driver with the same source `p` and target_signal `s` in the current simulation cycle. The relationship is between `d.waveform, d.waveform_extension` and `d'.waveform`.

The relationship shows which properties `d'.waveform value_transactions` must obey so that `d'.waveform` is a right successor of `d.waveform + d.waveform_extension`.

Note that for "computing" the current waveform the previous waveform_extension is used. Indeed, the time location of the current cycle can be determined only if the current waveforms are stable (ie. they do not change during the cycle). Therefore the waveform_extensions cannot be used in the same cycle they were produced in to modify the local (in time) waveforms.

Additional relationships are:

- For each process_driver the current_value depends on the current_value the driver had in the previous simulation cycle and the earliest pending driver value_transaction.

- For each signal_driver the current_value is the effective value of its source signal.

```
for each signal_status s IN current_cycle.signals
  for each d IN s.drivers

d.current_value =
    if is_type_of(d,'signal_driver')
    then s'.effective_value
         where s' IN current_cycle.signals
               such that s'.target_signal = d.source
    else if s.active_signal and
            earliest_time(d.waveform) = current_cycle.event_time
         then earliest_value(d.waveform)
         else d'.current_value
              where s' IN previous_cycle.signals and d' IN s'.drivers,
                    is_type_of(d','process_driver'),
              such that s'.target_signal = s.target_signal and
                        d'.source = d.source (same driving process)
```

3.4 Waveform relationships

This subsection deals with the difficult relationship between the previous and the current waveforms of a process_driver. It is assumed that waveforms are related to a fixed source and target_signal. For simplicity, it is also assumed that a waveform contains projected value_transactions (pairs) of the form (signal_value, occurrence_time) which can possibly occur in the future. The waveform is modified when new value_ transactions are generated as effect of executing signal assignment statements thus building up a waveform_extension.

Since the VHDL model is a static view of a dynamic process the execution of an assignment statement is modelled by means of an assign_transaction. An assign_transaction acts as a father of one or more value_transactions modelling in this way the cause - effect relationship between assignments and value_transactions. If we stick to the sequential semantics of VHDL, the modifications of a signal waveform are sequenced according to the sequence of contributing assignment statements executed in the same simulation cycle. A major decision is about the way the VHDL model captures this sequencing. There are two main possibilities:

- An assign_transaction must be the effect of executing a process which just become active due to a satisfied wait_transaction:

```
assign_transaction\source IN processes, where
processes= U swt.source
      swt IN satisfied_waits

(3) previous_wait_transactions +
    previous_satisfied_wait_transactions ->
    current_wait_transactions
```

- For each new wait_transaction (nwt), in the set

```
current_cycle.wait_transactions - previous_cycle.wait_transactions,
```

there must be at exist a satisfied_wait (swt) in the previous_wait_transactions such that swt.source = nwt.source. Therefore both transactions have the same source process. The swt is the cause of nwt.

- The set of satisfied_waits from the previous simulation_cycle is not included in the set of wait_transactions from the current simulation_cycle.

- The non-satisfied wait_transactions from previous simulation_cycle are included in the set of current simulation_cycle wait_transactions.

```
(4) previous_effective_value + current_drivers ->
   current_effective_value
```

- The effective_value of a signal_status depends on value the signal_status had in the previous simulation_cycle and on the current drivers of the signal_status. It also depends on whether or not the signal is resolved.

```
for each signal_status s IN current_cycle.signal_states
s.effective_value =
    if s.active_signal and -- there are pending waveforms
       s.earliest_value.time = current_cycle.event_time
    then
         if target_signal.resolved
         then s.resolving_transaction.effective_result
         else s.earliest_value.projected_value
    else s'.effective_value
         where s' IN previous_cycle.signals
         such that s'.target_signal = s.target_signal
```

where the earliest_value is the earliest scheduled value_transaction out of the current sets of signal waveforms.

```
(5) previous_process_driver_waveform +
    previous_process_driver_waveform_extension ->
    current_process_driver_waveform
```

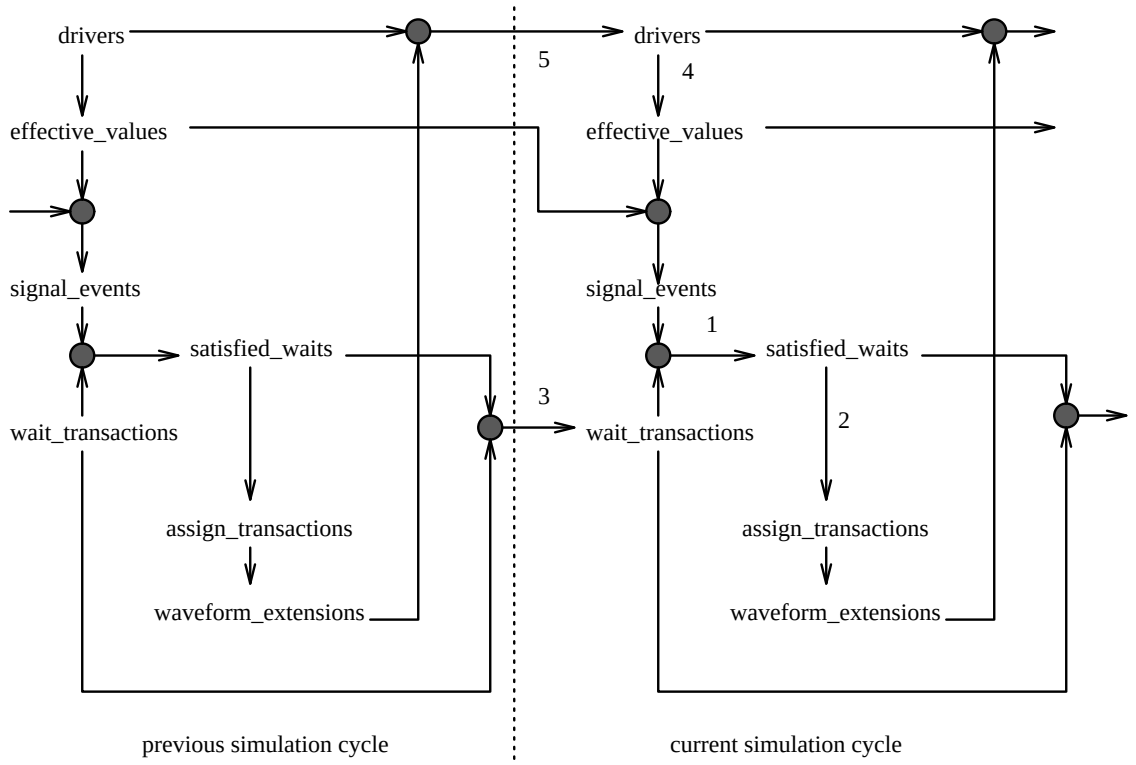

Figure 7: Major behavioural relationships

3.3.3 Concept relationships at the simulation level

The basic relationships between the concepts at the simulation level are illustrated schematically in figure 7. As shown, there are two kinds of relationships: inter-cycle and intra-cycle (local) all of them being more or less cause - effect dependencies. Obviously, there are additional dependencies at other model levels. In what follows the formulae like `a -> b` stand for a cause-effect relationship between `a` and `b`, ie. `b is the effect of a`.

A simulation_cycle starts either:

- at the earliest due_time of pending value_transactions (out of the overall projected set of value_transactions) or
- at the earliest due_time of pending delay_wait_transactions.

- For any wait_transaction there must exist exactly one source process in the simulated device_architecture.

```
(1) current_signal_events -> current_satisfied_signal_wait_transactions
```

- Each satisfied signal_wait_transaction (`swt`) must have a cause which is a signal_status `ss` such that the effective_value of `ss` has changed and

```
ss.target_signal IN swt.wait_for_signals
```

```
(2) current_satisfied_wait_transactions -> current_assign_transactions
```

3.3.2 Concept inter-relationships at the behavioural level

- The cause of a process_transaction is the execution of a process (source) which is member of the simulated device_architecture:

```
process_transaction\source IN
process_transaction\cycle.architecture.processes
```

Therefore, the device_architecture the process_transaction belongs indirectly to is the simulated device_architecture.

- The target_signal of an assign_transaction must be in the set of controlled signals of the process which generated the assign_transaction:

```
assign_transaction\target_signal IN
assign_transaction\source.controlled_signals
```

- The source of an assign_transaction (a process) must be member of the set of process sources of the assign target_signal:

```
assign_transaction\source IN
assign_transaction\target_signal.process_sources
```

- The signals a signal_wait_transaction waits for must belong to the sensitivity_signals of the process which caused the transaction to occur:

```
signal_wait_transaction\wait_for_signals <=
signal_wait_transaction\source.sensitivity_signals
```

- A delay_wait_transaction must occur in the current simulation cycle or in a future cycle:

```
is_greater(delay_wait_transaction\due_time,
            delay_wait_transaction\cycle.event_time)
```

- The due_time of a value_transaction (the effect of a "father" assign_transaction) must be greater than the current simulation_cycle time:

```
is_greater(value_transaction\due_time,
           value_transaction\father.cycle.event_time)
```

- For each process_driver/signal_driver of a target_signal component the process/signal which drives the component must belong to same device_architecture as the target_signal. The target_signal is an indirect attribute of the process_driver/ signal_driver, seen through the "owner" of the process_driver which is a signal_status concept.

```
driver\source IN
driver\owner.target_signal.architecture.processes
```

where driver stands for either process_driver or signal_driver.

For the dynamic concepts (such as process_transactions and signal_states) there are relationships between those concepts and the simulation_cycle they are scheduled to occur at. These relationships are either explicit or implicit (specified by means of attributes with special properties)

3.3 Basic relationships between the model concepts

Concept relationships can be categorised in the following major classes:

- `static relationships` (usually membership and type consistency dependencies);
- `cause - effect relationships`, which help modelling the behaviour.
- `time ordering relationships`, which anchor the cause - effect dependencies.

Usually, static relationships are concentrated at the structure level, while cause - effect and time ordering relationships are found at the behaviour and simulation levels.

It is important to note that there are implicit relationships specified via the concepts' attributes. For example, a value_transaction cannot exist unless it appears as a member in a set of value_transactions of the value_transaction father which is an assign_transaction. This implicit relationship is a cause - effect one. It says that a value_transaction is the effect of an assign_transaction. In what follows only the most important explicit relationships are mentioned. Type oriented relationships are completely omitted. They appear only in the outline model given in section 4.

3.3.1 Concept relationships at the structure level

The relationships between the structure level concepts address mainly the device_architecture membership. The problem is to make sure that all pieces put together to build a new concept are the right ones from the simulated device point of view.

- All signals used by a process must belong to the device_architecture which owns the process. Note that in the model a device_architecture contains, besides the local signals and local component ports, the ports of the device the architecture belongs to.

```
process\all_signals <= process\architecture.signals
```

- If a process controls a signal then the process must be a source of that signal:

```
process IN process\controlled_signal.sources
```

- In a connection both the source and the target signals must belong to the same device_architecture:

```
connection\source IN  connection\architecture
connection\target IN connection\architecture
```

- In addition, the source signal of a connection must be a member of the signal_sources of the target signal:

```
connection\source IN connection\target.signal_sources
```

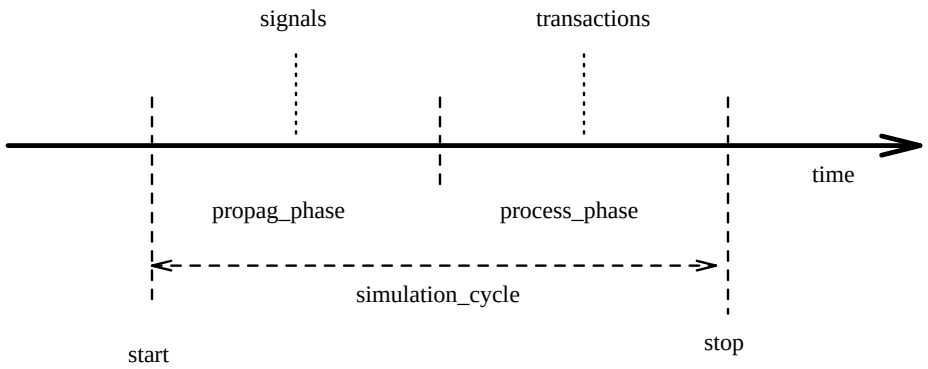

Figure 5: The structure of a simulation cycle

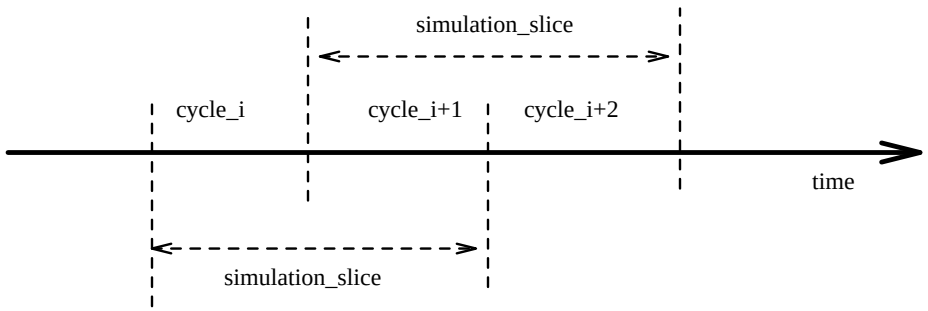

Figure 6: Simulation slices

- `satisfied_waits` - signal_wait_transactions and delay_transactions just satisfied in the current simulation_cycle;
- `non satisfied wait_transactions`, which will propagate to the next simulation cycle (together with the newly produced wait_transactions);
- `assign_transactions` - generated in the current simulation cycle as the effect of resuming the processes which have been waiting for the satisfied_waits.

- `Simulation_slice` - is a pair of time overlapping events (cycles),as shown is figure 6. It bundles the current simulation_cycle and the previous simulation_cycle. The simulation_slice is an artificial concept to help expressing inter-cycle concepts' relationships. For example, the cause - effect relationship between previous satisfied_waits and newly generated wait_transactions.
- `Simulation` - is a set of well behaved (in time) simulation slices. The simulation concept is used to capture the flow of the VHDL simulation process.

  - a signal_driver - if the component source is another signal, the component is connected to.

  A process_driver clusters:

  - the current signal component waveform (made up of a set of value_transactions generated in previous simulation cycles);
  - the waveform_extension, made up of value_transactions generated as effect of assign_transactions in the current simulation cycle.
  - the current component value.

  A signal_driver contains only the effective value of the source signal (value is considered, rightfully, inferred from the source signal status).

- `Signal_status` - designates the status of a signal in the current simulation cycle. It is characterised by the:

  - the signal_status simulation_cycle;
  - the target_signal;
  - the signal drivers (one for each signal source);
  - the effective_value of the signal (at the current cycle);
  - the signal_event.

  It said that there is an event on the signal if the effective_value of the target_signal really has changed from the previous simulation_cycle.

- `Resolving_transaction` - stands for the action of calling the resolving function for a resolved_signal. The result is not important. Only the relationship between the resolved signal effective_value and the value of the resolving_transaction matters. In turn, the relationship between the resolving computation_template and the actual call parameters in the resolving_transaction (from the typing point of view) is also important.

3.2.3 Concepts at the simulation level

As already stated, is assumed that VHDL behaviour simulation precess is divided into simulation cycles, as shown in figure 5. Cycles are successive in time. Each cycle contains two phases, from which, the first one - the "signal propagation phase" - can be missing.

The propag_phase has the aim to force new effective values on signals according to the projected waveforms and to infer signal_events (which signals have really changed their values). The process_phase is meant to resume processes waiting for some signal/condition events or delayed processes to be fulfilled. The model assumes that the propag_phase is done instantaneously at the start of the current cycle and the processing of wait_transactions is concurrent.

- `Simulation_cycle` - is an event which clusters:

  - cycle process_transactions;
  - signal states.

  In turn, the process_transactions, can be subdivided into:

- `assign_transaction` - represents the effect of executing a signal assignment statement from an active process. It easy to see that a process can generate several assign_transactions for the same simulation_cycle. An assign_transaction is characterised by a:
  - target_signal;
  - source (the process which executed the assignment statement);
  - set of value_transactions;
  - sequence_no;
  - delay mode.

  The sequence number of an assign_transaction is necessary to capture the order the signal assign statements were executed by their owner process. The sequence_no of an assign_transaction plays an important role in modelling the contribution the assign_transaction is supposed to make to the value of the target_signal.

  The delay mode (one out of transport and inertial) fixes the rules by which the contribution of the assign_transaction will take effect.

  The association signal - value_transaction is one to many from two different points of view. First a signal, if it is resolved, has several simultaneous sources which produce at least a value_transaction for each source. Second, each source can produce several pending values (transactions) in the same simulation_cycle, if the process generates several assign_transactions for the same target_signal. In order to classify the value_transactions in a neat way the assign_transaction is tightly related to the concept of (signal) "driver" explained in the sequel.

- `Value_transaction` - is a tuple

  ```
  (target_signal, source, signal_value, occurring_time)
  ```

  which designates one of the values the source-component of the target_signal might have in a future simulation_cycle. Note that for the same component there can be several projected values. Therefore each target_signal component (driven by a source) is characterised by its current_value and a projected waveform. The projected waveform has to be generated, according to specific rules, from the current component waveform and the waveform_extension, given by the set of new value_transactions produced by executing signal assignment statements.

  ```
  waveform= U (target_signal, source, signal_value, occurring_time)
          source= fixed
          target_signal= fixed
  ```

  The relationship between the current target_signal waveform and the new signal waveform, for each source driven component of the signal, is of major importance for the modelling of VHDL. A discussion of that matter makes the object of the next section.

- `Driver` - is an artificial concept which stands for the current and future status of a signal component (the only one or one out of many if the signal is resolved). A driver can be:
  - a process driver - if the component source is a process, and

- `assign_transactions` - actions which result in the modification of signals values;
- `wait_transactions` - actions which correspond to the suspension of a process either for a given amount of time or until a certain condition is met (for example, a signal changes its value);

By means of transactions a process can control a set of signals (controlled_ signals) and can depend on some other signals (sensitivity_signals).

- `Resolving computation_template` - is an important concept in VHDL which can also be placed at the structural level, although it is an abstract object and its semantics are not time dependent. In VHDL a signal can receive several values from different sources at the same moment in time. In that case the signal is said to be "resolved". The signal value at that time is computed by a resolving_function associated with the signal. From the modelling point of view, the structure of the resolving_function is not important. What matters is the "calling_pattern" of such a function, which shows the type of the result (the effective signal value) and the nature of the parameters: the signal source associated with a given parameter and the type of the expected signal component generated by the source. A template of that kind can be used to specify the constraints which have to be met by the sources and the result of a resolved signal.

3.2.2 Concepts at the behavioural level

The main concepts at this level designate actions happening in time and objects which can exist only during a given time interval. All behavioural transactions which exist at a given moment in time form an "event" which is an instance of the concept called simulation_cycle. The concepts clustered by means of simulation_cycles are:

- `Process_transaction` - a transaction generated by a process in the current simulation_cycle. Two important properties are inherited by all the concepts deriving from the process_transaction:
  - the simulation_cycle when the transaction was generated;
  - the source of the transaction (the process which generated it).

  A process_transaction can be:
  - `delay_wait_transaction` - an action the effect of which is to suspend its source process for a given amount of time. Note that a delay_wait_transaction can be member of several simulation_cycles starting with the cycle in which the transaction is generated and ending with the cycle when the transaction ceases to exist, resuming the suspended process. The time lap between the starting and the ending simulation_cycle events is exactly the amount of transaction waiting time.
  - `signal_wait_transaction` - is similar to a delay_wait_transaction with the only difference that the waiting condition can be combined: for a set of wait_for_signals and/or a value "condition". The value condition is a process local condition which is time independent and thus does not need to be made explicit from the modelling point of view. Similarly, a signal_wait_transaction can live through several simulation cycles.

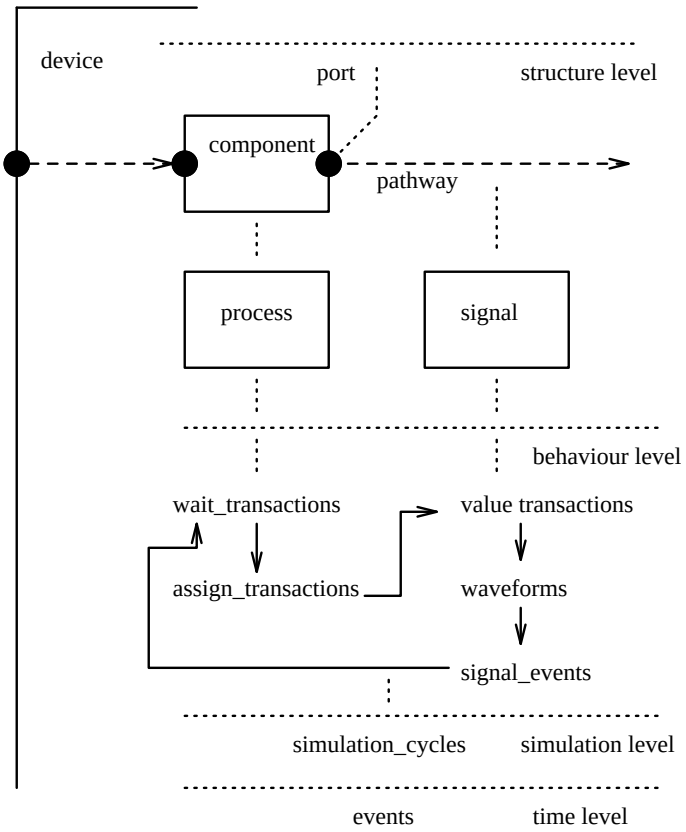

Figure 4: Levels of VHDL modelling

3.2.1 Concepts at the structure level

The following enumeration of the model main concepts at various modelling layers will help in understanding the role of the static and dynamic model objects.

- `Device` - stands for a hardware device provided with ports and the behaviour of which is described by a set of possible "device_architectures".
- `Device_architecture` - stands for the behavioural description of a device. An architecture uses: signals, components, processes and connections.
- `Signal` - designates an electrical value available at a device port or generated by a process. Therefore signals are classified into port_signals (or simply ports) and local_signals. The set of signals of a device_architecture includes, besides the component ports and local_signals, the ports of the device the behaviour corresponds to.
- `Component` - designates an instance of device being used locally within a device_architecture.
- `Connection` - stands for the connection of two signals (either local or ports) according to specific VHDL rules.
- `Process` - is a generator of transactions which specify the values of signals and the overall current status of the device. A process can produce:

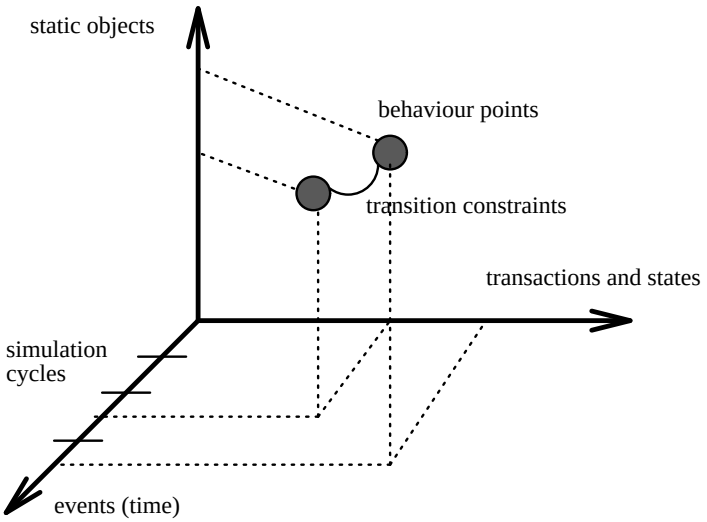

Figure 3: The VHDL modelling space

device. A wait_transaction is the effect of 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.

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, for example when the signals they wait for change 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.

Successive simulation cycles are paired in simulation slices. The `simulation slice` is taken as a base to specify constraints between wait_transactions in two successive cycles which, in turn, are considered as specific events.

3.2 VHDL modelling layers

The VHDL modelling spans across several layers starting from the VHDL data types and structure of a design and ending at the time model layer. The actual behaviour model is in between structure and time layers and encompasses the behaviour level and the simulation level as shown in figure 4. It is interesting that the type and time levels act as clustering referentials of concept instances on the structure, behaviour and, respectively, simulation level. In a structure, ports and signals are connected observing type constraints. Similarly, simulation cycles are ordered observing time constraints.

Thinking of the model in terms of layers helps not only to control inherent complexity but, it structures the way concepts are referenced.

At each layer the model captures relationships between VHDL underlying concepts or specifies local constraints at the level of each concept on the layer. The most important are concept relationships which link concepts at the same level or different levels of modelling.

3 Hints on VHDL behaviour modelling

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.

3.1 The basic approach of VHDL behaviour modelling

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.

```
signal s: integer:= 0;
P: process
   begin
      s <= (not s) after 20ns;
      wait for s;
   end_process P;
```

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:

- Concepts which stand for static (time independent) VHDL objects such as device, port, signal and process. 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 process and statement). A state represents the changing part of a VHDL static object. Therefore we can speak of, for example, process_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.

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

suspend events. For specifying this ordering, it is not necessary to measure time nor to restrict the duration of work_samples. In the limit, a work_sample can be completed instantaneously. Note, however, that there can be concurrent work_samples.

```
ENTITY work_sample;
activate,suspend: Event;

WHERE
  activate_then_suspend: is_after(suspend,activate);
END_ENTITY;

ENTITY process SUBTYPE OF (signal_source);
behaviour: SET OF work_sample;

WHERE
  no_time_interleave: check_interleave(behaviour);
END_ENTITY;

FUNCTION check_interleave (b: SET OF work_sample): BOOLEAN;
LOCAL i: INTEGER; END_LOCAL;

REPEAT i:= LOINDEX(b) TO HIINDEX(b) BY 1;
IF QUERY(w <* b | (is_after(w.activate,b[i].activate)   AND
                   is_before(w.activate,b[i].suspend))OR
                  (is_after(w.suspend,b[i].activate)    AND
                   is_before(w.suspend,b[i].suspend))) <> []
THEN  RETURN (FALSE);
END_IF;
END_REPEAT;

RETURN (TRUE);
END_FUNCTION;
```

Considering that the work samples contain information describing the status of a dynamic system, a model of the behaviour of that system can be built by stating the appropriate relationships between status information in different time-ordered work samples. That is the case for VHDL behaviour modelling, where simulation cycles play the role that work samples have in the example above.

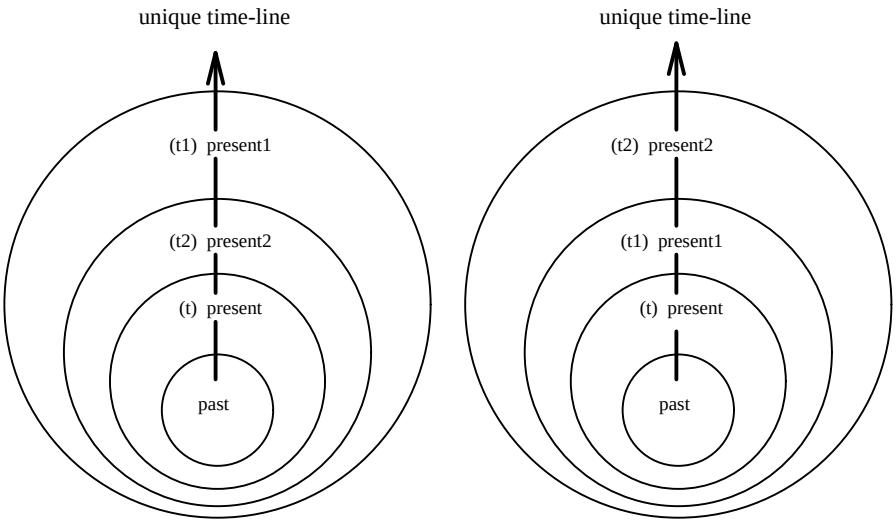

Figure 2: The basic time space

particular time point - called, say, the "big-bang" point - with an empty past and time equal to `time_start`.

In the above model, events which happen in no time can still be ordered. This is possible since the ordering is based on event set containment and not on the event_time values. Therefore two time points `t1` and `t2` can exist such that `t1.present + t1.past = t2.past, t1.time = t2.time`. Useful predicates can be defined accordingly to test if an event is next or prior to another event.

```
FUNCTION is_next(e1,e2: Event): BOOLEAN;
RETURN (e1.event_time_point.past =
       (e2.event_time_point.present + e2.event_time_point.past));
END_FUNCTION;

FUNCTION is_before(e1,e2: Event): BOOLEAN;
  RETURN ((e1.event_time_point.present + e1.event_time_point.past) <=
          e2.event_time_point.past);
END_FUNCTION;
```

Considering the linear time model, `Time_value`s can be used freely in constraints in the same way that distances or weights are used. Order relationships between events and the time location of events can now be specified by means of time values. Moreover a time point can be selected indirectly, for example relying on some properties of an entity such as the minimum response time to an input change.

2.4 Using time models

To show how the linear model of time can be used, suppose that there exists an entity `process` which contains a sequence of work samples, each sample being characterised by two events: `activate` and `suspend`. The `work_sample` sequence is correct if its elements cannot be interleaved. That implies chronological ordering of work samples according to the `activate` and

In addition, `time_start` is a constant standing for the minimum valid `event_time` according to the application which uses the time model. It is also assumed that `Time_value` measures an absolute time for all events and that basic operators, such as `is_equal` and `is_less`, are defined. These operators can hide `time_unit` and whatever conversions are necessary to work with time values. For convenience, the `Time_value` operators are used only within EXPRESS constructs. Elsewhere in the text and also in EXPRESS comments the operators `=, <>, <, >, -, +` will be used instead.

```
ENTITY Event;
event_time_point: Time_point;
DERIVE
event_time: Time_value := event_time_point.time;
END_ENTITY;

ENTITY Time_point;
past: SET OF Event;
present: SET [1: ?] OF Event;
time: Time_value;

WHERE
valid_time_value:
   is_equal(time,time_start) OR
   (past <> [] AND is_greater(time,time_start));

directional_time: present * past = [];

event_ordering:
  QUERY(e <* past | is_less(e.event_time,time) OR
                    is_equal(e.event_time,time)) = past;
END_ENTITY;

RULE linearise_time FOR (Time_point);
-- for t,tp IN Time_point either is_after(t,tp) or is_after(tp,t)
WHERE
QUERY(t <* Time_point |
      QUERY(tp <* Time_point | (t = tp) OR
                               ((t.past + t.present)<= tp.past)OR
                               ((tp.past + tp.present)<= t.past))
      = Time_point)
= Time_point;
END_RULE;

RULE unique_present FOR(Time_point,Event);
END_RULE;  -- See section 2.2
```

In can be proved that the model above obeys definition 2.1. The time point space contains a single `time-line` as illustrated in figure 2. According to the event_ordering constraint, the total ordering of time points preserves the event_time ordering. In addition, there exists a

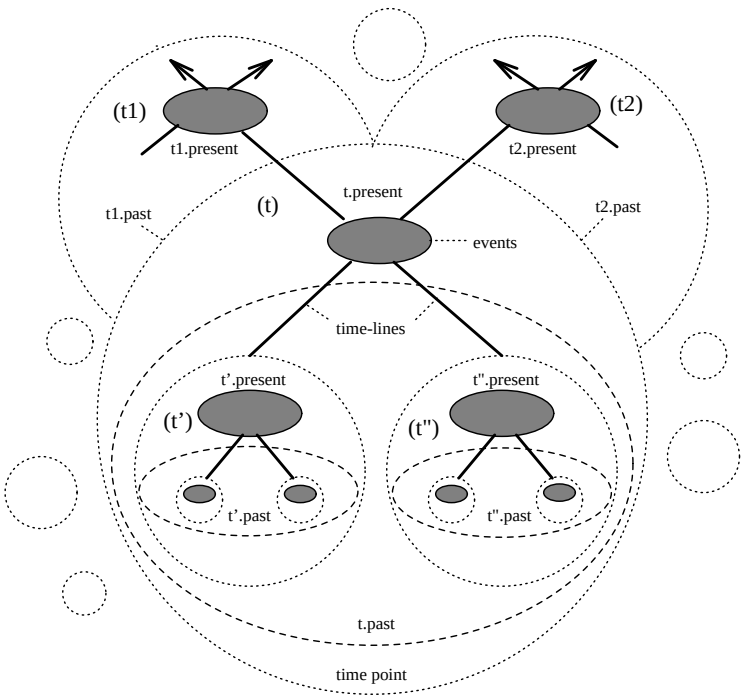

Figure 1: The basic time space

same time-line (`is_after` defines a partial order on `Event`). Linearising the `Time_point` space is, therefore, useful.

`Definition 2.1.` A time point space is considered linear if for any two points `t1, t2 IN Time_point, t1 <> t2`, and for any two events `e1 IN t1.present, e2 IN t2.present` either `is_after(e1, e2)` or `is_after(e2, e1)`.

In a linear time point space the `is_after` operator orders totally the `Event` set. Although the `Time_point` space can be linearised without introducing a time measure, as shown in [GC92], in what follows only a measurable linear variant of the time space is presented.

2.3 Measurable time

In many applications time lapse, used to express the duration of events, can be important. In order to be able to measure time the basic time model is updated by introducing a "value of time". The entity `Time_value` is intended for that purpose. `Time_value` can be defined as a covering of application dependent time measuring units which, in turn, are defined in terms of a basic `time_unit`, provided that the set of `time_unit`s is totally ordered.

```
ENTITY Time_value ABSTRACT SUPERTYPE;
END_ENTITY;

CONSTANT time_start: Time_value:=
        <application dependent constant>;
END_CONSTANT;
```

```
ENTITY Time_point;
past: SET OF Event;
present: SET [1 : ?] OF Event;
-- present cannot be empty

WHERE
directional_time: present * past = [];
-- time flows by

nothing_is_forgotten:
  QUERY(e <* past | (e.event_time_point.past +
                     e.event_time_point.present) <= past) = past;

-- SELF\.past contains all events
-- which are on time-lines entering SELF
END_ENTITY;

RULE unique_present FOR(Time_point,Event);
-- each event must be in the present
-- of its associated Time_point
WHERE
QUERY(t <* Time_point | QUERY(e<* Event |e.event_time_point:=:t)
                        = t.present)
= Time_point;
END_RULE;
```

Although the model does not specify any metric for time measuring purposes, it makes it possible to define operators such as `is_after` and `is_before`, and therefore to order the events. For example,

```
FUNCTION is_after(e1,e2: Event): BOOLEAN;
-- Checks if e1 occurred after e2

RETURN((e2.event_time_point.present + e2.event_time_point.past) <=
        e1.event_time_point.past);
END_FUNCTION;
```

According to the model above, the structure of a time point space is as illustrated in figure 1, where `t,t1, and t2` are `Time_points` such that

```
t.past + t.present <= t1.past,
t.past + t.present <= t2.past,
t1.present * t2.present = [].
```

Several `time-lines` can emerge from the same present. Moreover, several distinct `time-lines` (chains of time points) can coexist in the same past. Therefore, the time point space contains parallel time point subspaces which, eventually, can be disjoint.

Although these properties are general, they make it more difficult to test the ordering of events. For instance `is_after(e1,e2)` makes sense only if the events `e1` and `e2` are on the

these attempts fall short of the basic aim of modelling: to describe without computing. In addition, there are external factors such as the way of evaluating formal relations between model entities, which reintroduce time through the back door.

The reasons above suggest that time values are an insecure basis for time modelling. Instead, the ordering of events according to their existence, regardless of their occurrence time, is essential. The time values can be thought of as a consequence of event ordering and, therefore, the time model will be self contained and "exact".

2.2 The basic time model

Time, as an abstract concept, is intended for event sequencing. It designates a space of time points reached from one another by "before" and "after"-like operators. For humans, time is also understood in terms of events. We remember events not time. From this point of view it can be said that time has a corpuscular nature; quanta are events. Nevertheless, time is passing by, structuring the time point space in a way which makes it possible for an observer to orient in time by distinguishing the chronological ordering of events, regardless of the time distance between events. From this point of view time is continuous.

Considering events as fundamental and viewing time as a space, `Event x Time_point` comes near to the temporal logic proposed by Kowalski and Sergot [KS86]. However, the structure of this space is made explicit here rather than through being used to build logic formulae.

Set theory is used in what follows as it is not only a convenient tool for expressing the concepts discussed, but it also provides operations suitable to capture the essence of relations underlying the semantics of time. A particular `Time_point` contains the sets of events which are happenning (the present events) or have happened (the past events). In this way the concepts "past" and "present" themselves can also be defined in terms of sets of events which can be ordered on a set inclusion basis.

Let `Event` be the generic set of events and `Time_point` be the set of time points. The following essential constraints have to be observed:

- For each `t IN Time_point, t` contains two sets of events called `t.present` and `t.past`. It is considered that when the events in `t.past` were created the events in `t.present` did not exist.
- For each `e IN Event` there is a unique point `t IN Time_point` such that `e IN t.present`. It is said that `e` happens (is created) at time point `t` (denoted by `e.event_time_point`).
- For each `t IN Time_point` and `e IN t.past`, all predecessors of `e` and also all events created at `e.event_time_point` are in `t.past`. Note that the predecessor relationship is relative to those `time-lines` which come from the past and pass through `e.event_time_point`.

These constraints are satisfied by the EXPRESS model below which uses the following set operators: `IN` for set membership, `+, *, -` for set union, intersection and difference, `<=` for set inclusion, `=` and `<>` for set equality and inequality, and `[]` for representing the empty set.

```
ENTITY Event;
event_time_point: Time_point;
END_ENTITY;
```

1 Introduction

In contrast to computation, which aims to show how values which characterise a universe of discourse are obtained, modelling is intended to capture the conceptual structure of the universe: the underlying concepts, their relationships and constraints. The differences between computation and modelling can be found at lower levels in computer paradigms; for example conventional versus knowledge oriented programming, imperative versus declarative programming. The main thrust of computation is to solve. The main effort in modelling is to describe. This goal is of significance in CAD especially for supporting the transfer of design between different CAD systems and for design verification. As a result interest in developing modelling languages and in modelling various CAD domains is growing.

This report addresses some particular cases in that area: time modelling in general, time modelling in EXPRESS [ISO91], and time-based behaviour modelling in hardware design, particularly for VHDL [LSU89]. Its main purpose is to propose a way time-dependent behaviour, in particular VHDL behaviour, can be modelled by using simple conceptual relationships. In addition, it presents an outline model of VHDL behaviour in EXPRESS using the existing language features.

2 A model of Time

This section presents variants of a time model relying on time space structure rather than on numerical values used to measure time. The model observes common sense time properties and can be used in modelling time based behaviour of various artifacts. The ability to provide a generic model of time is essential to the use of information modelling of domains such as VHDL. The model must conveniently describe time sequencing of instantaneous events. This is a must for modelling VHDL since, due to delta delays, there can be successive simulation cycles with the same time value.

Only two time models are presented here. A more comprehensive discussion on EXPRESS time modelling can be found in [GC92].

2.1 Time values and events

Extensive work has been carried out to develop formal tools suitable for describing behaviour [SMW92]. Temporal logic is an example. These approaches tend to consider temporal precedence relations as primitive concepts. This assumption would be impossible when using a modelling language, such as EXPRESS for example, where all concepts and relationships have to be made explicit. In modelling languages, the meaning of, for example, `is_after` or `is_before` must be described in terms of more basic concepts, concepts which actually model time.

In addition, it is inadequate to assume that a set of numerical values, call them `time values`, can represent a model of time. Philosophically, the restrictions of using numerical values when these are meant to tell time are in the mind of the user instead of being made explicit in the model [GA78]. Even supposing that all necessary restrictions could be made explicit, the numeric time model is still debatable from the point of view of the absolute measuring precision required.

There are attempts [WD92] [VT92] to model behaviour which try to avoid the above pitfalls by using operators which compute the status of the modelled universe at the "next" moment in time and then specify relationships between the current and future universe states. However,

This report addresses time-based behaviour modelling in hardware design, particularly for VHDL. Its main purpose is to propose a way time-dependent behaviour, in particular VHDL behaviour, can be modelled by using simple conceptual relationships. In addition, it presents an outline model of VHDL behaviour in EXPRESS.

Contents

1 Introduction **2**

2 A model of Time **2**
2.1 Time values and events 2
2.2 The basic time model 3
2.3 Measurable time 5
2.4 Using time models 7

3 Hints on VHDL behaviour modelling **9**
3.1 The basic approach of VHDL behaviour modelling 9
3.2 VHDL modelling layers 10
3.2.1 Concepts at the structure level 11
3.2.2 Concepts at the behavioural level 12
3.2.3 Concepts at the simulation level 14
3.3 Basic relationships between the model concepts 16
3.3.1 Concept relationships at the structure level 16
3.3.2 Concept inter-relationships at the behavioural level 17
3.3.3 Concept relationships at the simulation level 18
3.4 Waveform relationships 20
3.5 Informal specification of the waveform relationship 23
3.5.1 The waveform constraint (C_SD) 23
3.5.2 An example 25
3.6 Additional restrictions on signal_waveforms and value_transactions 26

4 The VHDL outline model **29**
4.1 Structure level model 29
4.2 Behaviour level model 35
4.3 Simulation level model 41
4.4 Time level model 46

5 Conclusions **49**

6 Acknowledgements **49**

ESPRIT Project 2072 ECIP2

European CAD Integration Project

Prepared by

Cristian A. Giumale (University of Manchester),

Towards a VHDL Model
Preliminary Report

ECIP2/UM/008-1 22-10-92

©Copyright 1992 Bull S.A., International Computers Ltd., Siemens Nixdorf Informationssysteme AG, Nederlandse Philips Bedrijven B.V., Racal-Redac Ltd., Siemens AG, Thomson-CSF, University of Manchester, Paderborn University. This document and the information contained herein may not be copied, used or disclosed in whole or in part except with prior written permission of the partners as listed above. The copyright and the foregoing restriction on copying, use and disclosure extend to all media in which this information may be embodied, including magnetic storage, computer print-out, visual display, etc. ... The document is supplied without liability for errors or omissions.