ESPRIT Project 2072 **ECIP2**

European CAD Integration Project

Prepared by

Cristian A. Giumale (University of Manchester),

Models of Time
Revised Report

ECIP2/UM/009-5 **29-01-93**

Abstract

The report presents several 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 artefacts. Discrete versus continuous time modelling is also discussed and a way of modelling continuous-time behaviour is proposed. The ability to provide a fundamental model of time is essential to the use of information modelling of domains such as VHDL.

Contents

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 discusses some specific cases in this area: time modelling in general, time modelling in EXPRESS [ISO91], and time-based behaviour modelling in hardware design. It presents time models which are suitable for describing event sequencing which is, eventually but not necessarily, measurable.

In a static universe, conceptual relationships, whatever particular form they might have, are basically structural. However, many applications use time dependent universes of discourse. For example, VHDL [LSU89] [IE88] descriptions rely heavily on time. Modelling universes where time-based behavioural aspects are fundamental, requires the capacity to capture and describe time dependent conceptual relationships. The modelling task is simplified if such relationships can be expressed in the same way structural relationships are.

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, is_before` must be described in terms of more basic concepts, concepts which actually model time.

Superficially the task seems easy if time is thought of merely as a continuum of numeric values used to measure time. Let us call these `time values`. The relevance of such a model is debatable. Assume, for instance, that two causally related events `e1` and `e2` happen at two different points in space at time values `t1` and `t2`. If the events are ordered according to `t1` and `t2` that ordering depends on whether `t1` and `t2` are local or proper times, on the units for the time metric and on the precision of the time metric. In addition, it may happen that both `e1` and `e2` complete in zero time and $t1 = t2$, as is the case with "delta delay" signal assignments in VHDL [IE88] [LSU89].

It is inadequate to assume that a set of numerical 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 [WE78] [RF88].

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, 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'. Nevertheless, discussing time and, finally, modelling these time ordering relations, is impossible without adhering to some philosophical assumptions. In itself, the problem of event existence is a philosophical issue. In what follows, it is considered that events exist objectively and absolutely. Therefore, there exists an absolute arbiter which can tell if an event existed prior to another. In addition, it is assumed that time cannot exist outside the event space and that time is directional.

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`. 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;

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_previous`, 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

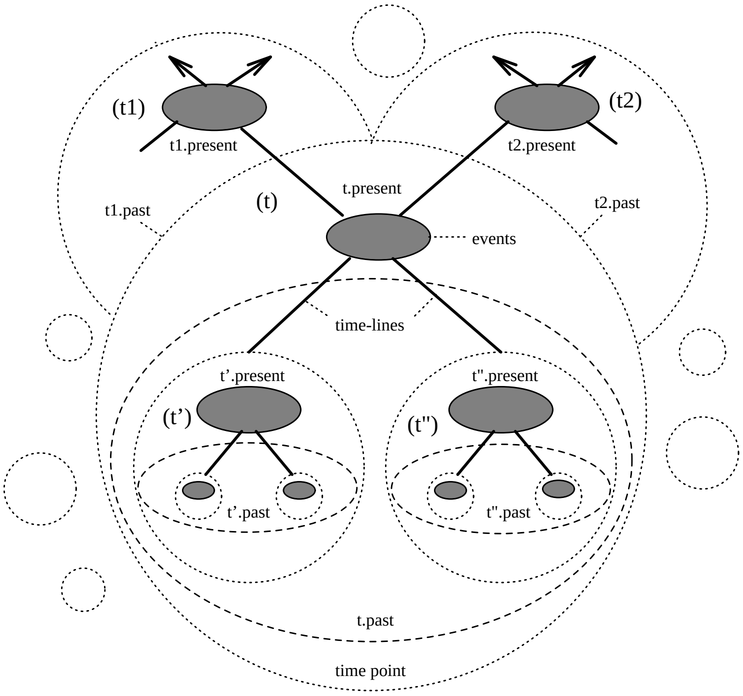

Figure 1: The basic time space

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

`Definition 1.` The time point space is considered linear if for any two points `t1, t2 IN Time_point, t1 <> t2`, and for any 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. The model has to be modified as below.

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

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

WHERE
  directional_time: present * past = [];
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 two points t, tp IN Time_point
--     t must be after tp or tp must be after 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;
```

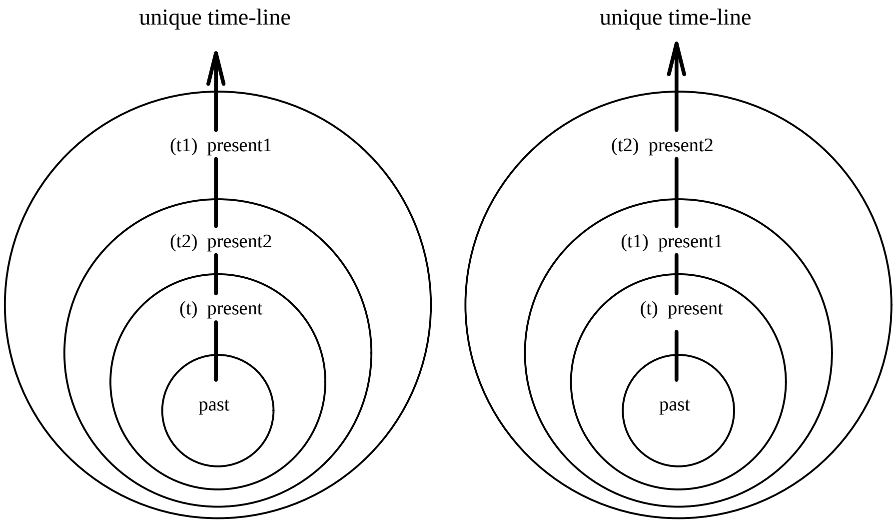

Figure 2: A linearised time space

`Proposition 1.` The rule linearise_time linearises the `Time_point` space.

Proof.

`Case a.` A `Time_point` space with a single element is linear.
`Case b.` Consider `Time_point` a linear space with `n` elements such that for two time points ti and ti+1

```
ti.past + ti.present <= ti+1.past, i= 1, n-1
```

and let `t, t <> ti, i= 1,n` be a new `Time_point`. According to the linearise_time rule only the following alternatives are possible

```
t.past + t.present <= t1.past   or
tn.past + tn.present <= t.past  or

ti.past + ti.present <= t.past,  i= 1, k, and
t.past + t.present <= tj.past,  j= k+1, n, for 1 <= k <= n
```

Therefore, any two events from the present of two different time points in the set `t1, ...,tn+1` are ordered according to `is_after`. By induction, it follows that the `Time_point` space is linear. A linear time space looks as illustrated in figure 2. The time point space contains a single `time-line`.

The linear time model has an important property: there exists a particular time point - called, say, the "big-bang" point - with an empty past.

`Proposition 2.`
There exists an unique point `t0 IN Time_point` with `t0.past =[]`. For any `t IN Time_point, t <> t0` then `t.past <>[]`.

Proof.

1) Let `t` be a time point such that `t.past <> []`. Therefore, for any `e' IN t.past` and `e IN t.present, is_after(e,e')` is true. Chose `e' IN t.past` and let `t'= e'.event_time_point`. From

```
t'.past + t'.present <= t.past,
t'.present <> []
```

it follows that `0 <= SIZEOF(t'.past) < SIZEOF(t.past)`. Since the `past` event set is finite, by induction on `past` size, it follows that `t0` exists.

2) Consider `t IN Time_point, t <> t0, t.past = []`. The linearise_time rule is contradicted, since neither of the following relations can hold

```
a) t0.past + t0.present <= t.past
b) t.past  + t.present  <= t0.past
```

Therefore `t.past` cannot be empty and t0 is unique.

3 A measurable time model

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_units` is totally ordered. In addition, there can exists, if necessary, a `time_start` constant standing for the minimum valid `event_time` according to the application which uses the time model.

```
ENTITY Time_value ABSTRACT SUPERTYPE;
END_ENTITY;

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

It is also assumed that `Time_value` measures an absolute time for all events in the `Event` generic set and that basic operators such as `is_equal`, `is_less`, `is_greater`, `time_diff` and `time_add` are defined. These operators can hide `time_unit` and whatever conversions are necessary to work with absolute 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;

UNIQUE time; -- time points must have different time values
WHERE
   valid_time_value: -- not essential
   is_equal(time,time_start) OR
   ((past <> []) AND is_greater(time,time_start));

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

  -- for any e IN past, e.event_time < time.
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);
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;
```

It is easy to see that, according to the event_ordering constraint, the total ordering of time points preserves the event_time ordering. In addition, the constraint valid_time_value forces the time value of the "big-bang" point to be `start_time`. Both the event_ordering constraint and the linearise_time rule imply the uniqueness of the time value of a `Time_point`. The UNIQUE clause has been added for clarity only.

The measurable time model can be further simplified by capturing in a single rule the total ordering of `Time_point` space according to the event_time.

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

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

RULE linearise_time FOR (Time_point, Event);
--  for any t IN Time_point and e IN Event
--      if e.event_time < t.time then e IN t.past

WHERE
  QUERY(t <* Time_point |
        QUERY(e <* Event | is_less(e.event_time, t.time)) = t.past)
  = Time_point;
END_RULE;
```

```
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;
```

Indeed, suppose that the time space is as in figure 1. For the new linearise_time rule to hold the following must be satisfied:

```
Case (a) t1.time < t2.time, then
         t.past + t.present <= t1.past
         t.past + t.present <= t2.past
         t1.past + t1.present <= t2.past

Case (b) t1.time > t2.time, then
         t.past + t.present <= t1.past
         t.past + t.present <= t2.past
         t2.past + t2.present <= t1.past
```

`Case (c) t1.time = t2.time` is impossible due to the time uniqueness constraint in the `Time_point` entity definition.

Therefore, any two events from different time points are surely related by `is_after` relation.

Considering the linear time model `Time_value`s can be used freely in constraints in the same way thet 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.

4 An instantaneous event time model

The linear time model from section 3 is based on common sense by assuming that events take some time to be completed. However, there are applications which require more. For example, in VHDL [IE88] the status of a signal can be changed instantly according to "delta delays". Therefore, an extended model of linear time able to cope with instantaneous events is necessary.

In this model events which happen in no time can still be ordered. This is possible since the ordering is based on the 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`. The main modification is to alter slightly the `event_ordering` constraint and to drop the time uniqueness clause from the `Time_point` entity. The `Event` entity remains unchanged.

```
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;

  -- 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);
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;
```

Note that since the `UNIQUE time` clause has been removed from the `Time_point` entity, and the time value of past points can equal the time value of present points, there can be a non empty set of time points `t` with `t.time = time_start` and with `t.past <> []`.

5 Time properties

How do we know that the above models are correct? The method used here is to formulate specific time properties and then prove that they are satisfied by the models. The following common properties have been chosen here:

- Events are unique in the sense that an event cannot simultaneously be both in the present and in the past of the same time point.
- Any point in the `Time_point` space, except the "big_bang" point, can be reached from any point in its past.
- Time is irreversible. The events in the past of a time point cannot "vanish" and cannot be "induced" from subsequent time points.

These properties are satisfied by all linear time models given above, as stated by the following propositions.

`Proposition 3.` Events are unique. That follows directly either from the directional_time constraint (in the linear but not measurable model and in the instantaneous event model) or from the event_ordering constraint (in the linear measurable model).

`Definition 2.` (The immediate successor of a time point) Assume `t,t' IN Time_point`. It is said that `t` is the immediate successor of `t', t = next(t')`, if `t'.past + t'.present = t.past`.

`Proposition 4.`
For any point `t IN Time_point, t.past <> []` there exists `t' IN Time_point` such that `t = next(t')`.

Proof. Assume `t'` does not exist. Choose any `tp IN Time_point` such that

```
tp.past + tp.present <= t.past   (a)
```

Such a point exists, since (1) `t.past <> []`; (2) according to the unique_present rule the events from `t.past` must be in the present of some time points which, (3) due to the linearise_time rule, must obey (a).

From the hypothesis it follows that

```
tp.past + tp.present <> t.past   (b), and from (a)
tp.past + tp.present < t.past    (c)
```

Since relation (c) holds for any predecessor of `t` it follows that there is at least one event which is not contained in the present of any time point. This is impossible since it contradicts the unique_present rule. Therefore `t'` exists.

`Proposition 5.`
Between any two points `t, t' IN Time_point`, `t'.past <> []` , `is_after(t', t)` there exists an unique path `t -next-> t'` of the form `t'= next o ... o next(t)`.

Proof.
`Case 1.a:` `t.past + t.present = t'.past`. Obviously `t'= next(t)`.

`Case 1.b:` `t.past + t.present < t'.past`, where `<` stands for the proper set inclusion. Consider `ti -next-> t', ti IN Time_point` with

```
t.past + t.present < ti.past            (a)
```

Choose `ti+1 IN Time_point` such that

```
ti+1.past + ti+1.present = ti.past      (b)
t.past + t.present <= ti+1.past         (c)
```

The point `ti+1` exists. Indeed, (b) is satisfied from proposition 4. If (c) is not satisfied, then according to the time linearise_time rule

```
ti+1.past + ti+1.present <= t.past      (d)
```

which from (b) and from `t.present <> []` gives `ti.past < t.past + t.present`, thus contradicting hypothesis (a). In addition, `ti` is unique because time points are unique. Therefore, the path `ti -next-> t'` can be extended to `ti+1 -next-> t'`. From (b), (c) and (a) it follows that

```
SIZEOF(ti+1.past - t.past - t.present)  <
SIZEOF(ti.past - t.past - t.present)
```

By induction on `i` there is an `i= n` such that

```
SIZEOF(tn.past - t.past - t.present) = 0, or
tn= next(t)
```

Therefore, `t -next-> t'` exists.

`Proposition 6.` Time is irreversible. Let be `t1, t2 IN Time_point` such that `t1= next(t2)`. Then there cannot exist `e IN Event` such that

```
((e IN t2.present) AND  NOT (e IN t1.past))
 -- events cannot vanish
OR
(NOT (e IN t2.present) AND NOT (e IN t2.past) AND (e IN t1.past))
 -- events cannot be induced
```

The proof is obvious, since the above relation violates the definition of `next` which, in turn, is based on the linearise_time rule.

6 Event ordering predicates

When modelling the behaviour of a given class of objects the model has to express event relationships. For that purpose tests such as `is_next, is_previous, is_concurrent, is_after, is_before` occur quite often. Apart from `is_after`, specified in section 2, the following functions implement the predicates mentioned above for the linear time models.

```
FUNCTION is_next(e1,e2: Event): BOOLEAN;
-- Checks if e1 is an immediate successor of e2

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;
-- Checks if e1 is concurrent with e2

RETURN (e1.event_time_point.present = e2.event_time_point.present);
END_FUNCTION;

FUNCTION is_previous(e1,e2: Event): BOOLEAN;
-- Checks if e1 is an immediate predecessor of e2

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;
-- Checks if e1 is a predecessor of e2

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

Note that the functions above are based on event set containment and not on event_time values. Therefore, they can be correctly applied for the instantaneous event time model.

7 Using time models

To show how a linear model of time can be used (any linear model), 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 the `work_samples` are not interleaved. That implies chronological ordering of work samples according to the `activate` and `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.

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.

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 nanoseconds. After each signal assignment `s <= (not s) after 20ns` the process `P` is blocked for 20ns waiting for `s` to change its value.

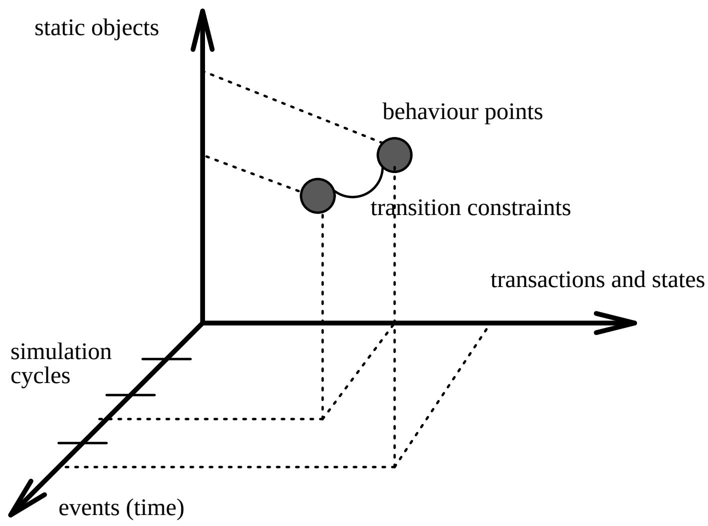

Figure 3: The VHDL modelling space

```
   signal s: bit:= '0';
P: process
   begin
      s <= (not s) after 20 ns;   (a)
      wait on 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 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 slice is taken as a base to specify constraints between wait_transactions in two successive cycles which, in turn, are considered as specific events.

The model below is a simplified outline of `cause-effect` dependency between wait_transactions generated by processes belonging to a device.

```
ENTITY process_transaction
       ABSTRACT SUPERTYPE OF (ONEOF (wait_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 device 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;
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 simulation_cycle SUBTYPE OF (Event);
  architecture: device_architecture;
  wait_transactions: SET OF wait_transaction;
  ...

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

  satisfied_waits: SET OF wait_transaction:=
        select_satisfied_waits(wait_transactions, cycle);
  -- wait_transactions the wait condition of which is satisfied

 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_transactions are satisfied. It
  -- depends on which from the above mentioned events come 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
  ...
END_ENTITY;

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

DERIVE
  prev_cycle: simulation_cycle:= NVL(previous, init_cycle);
  -- prev_cycle:= IF EXISTS(previous) THEN previous ELSE init_cycle
  -- init_cycle is a constant for the initial state of simulation

  new_waits: SET OF wait_transaction:=
  current.wait_transactions - prev_cycle.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);

  new_wait_transactions_constraint:
  QUERY(new_wt <* new_waits |
        SIZEOF(QUERY(wt <* prev_cycle.satisfied_waits |
                     wt.source :=: new_wt.source)) = 1))
  = new_waits;
  -- a new wait transaction must have a cause which is a previous
  -- wait satisfied transaction from the same process

  old_satisfied_waits_are_discarded:
  prev_cycle.transactions.satisfied_waits *
  current.transactions.wait_transactions = [];

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

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

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

  simulation_membership:
  QUERY(sl <* simulation | sl.architecture :<>: architecture) = [];
END_ENTITY;
```

Note that the model above can cope with `delta delays`, if the instantaneous event model of time is used. For example, if the assignment on line (a) from the process `P` is `s <= (not s) after 0`, all simulation cycles have the same `event_time`. Nevertheless, they are correctly ordered according to the time model.

Other relationships between time-dependent VHDL concepts can be specified in a way similar to that shown above.

8 Time-continuous versus time-discrete behaviour

It can be argued that the time models presented are able to cope with discrete time-based behaviour only. However, adhering to the view that discrete behaviour can be taken as an approximation of continuous behaviour, these models can provide a basis for modelling `time-continuous` behaviour.

8.1 Modelling time-continuous behaviour

`Definition 3.` The behaviour of an object `Ob` is said to be epsilon "time-continuous" if for a chosen `epsilon IN REAL, 0 < epsilon << 1`, it is characterised by a finite, eventually unbounded, set `Ev(Ob)` of events and a morphism `T:Time_value -> REAL`, which satisfy

```
    for any e IN Ev(Ob) which has at least one predecessor in Ev(Ob),
        there exists e' IN Ev(Ob) such that
            is_after(e,e') AND |T(e.event_time - e'.event_time)| <= epsilon
```

`Definition 4.` The behaviour of an object is said to be `time-continuous` if it is epsilon time-continuous for any `epsilon IN REAL, epsilon > 0`.

The above definitions have been inspired from the mathematical definition of a continuous function `f: REAL -> REAL` by taking into account finite, although perhaps unbounded, sets of events. In addition, note that time continuous behaviour is a relaxed form of continuous behaviour as illustrated in figure 4.

A model which satisfies definition 3, and which does not fix the value of epsilon, could be regarded as being generic, and, according to definition 4, as describing the "ideal" continuous-time behaviour of a given artefact. It is a covering of an infinite set of instances of discrete behaviour models which approximate the described artefact.

As a simple example, consider the modelling a waveform `w= SIN(period * t + phase), t IN REAL, t >= 0`. First, the `Time_value` and the `Time_value` operators are as follows.

```
CONSTANT
  time_start: Time_value:= Time_value(0.0);
END_CONSTANT;

ENTITY Time_value;
  t: REAL;
  UNIQUE t;
END_ENTITY;

FUNCTION is_greater(ta,tb: Time_value):BOOLEAN;
  RETURN (ta.t > tb.t);
END_FUNCTION;
```

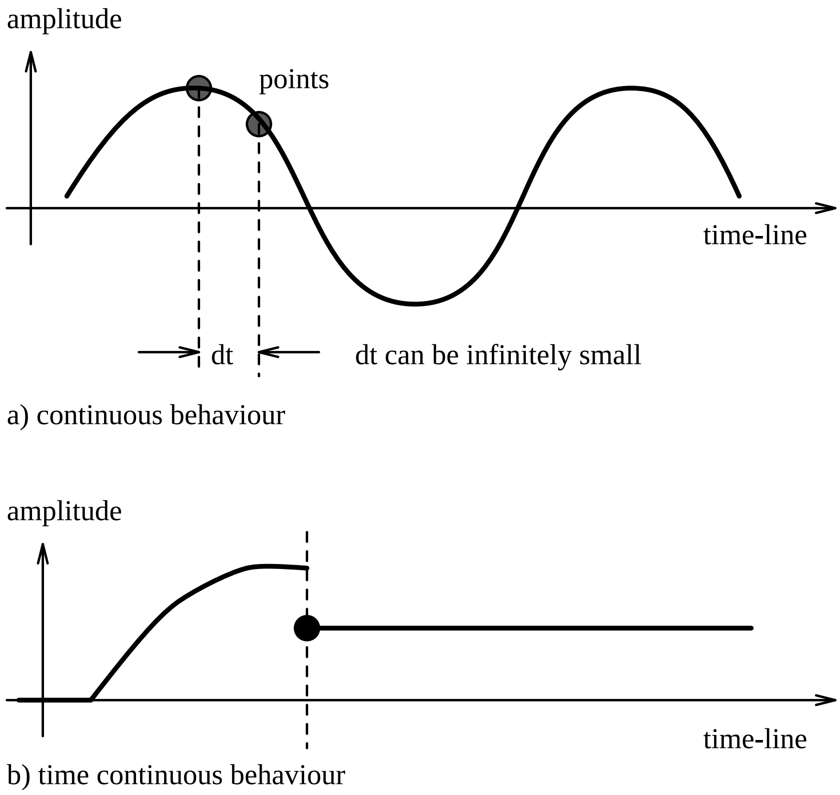

Figure 4: Time continuous behaviour

```
FUNCTION is_equal(ta,tb: Time_value):BOOLEAN;
  RETURN (ta.t = tb.t);
END_FUNCTION;
. . .
FUNCTION T(tav: Time_value):REAL;
  RETURN (tv.t);
END_FUNCTION;
```

Note that the function `T: Time_value -> REAL` exists. Moreover, `Time_value` represents a generic set which satisfies the following property:

```
    for an arbitrary chosen epsilon IN REAL, epsilon > 0      (a)
        Time_value can be instantiated such that
    for any tv IN Time_value
        there exists at least one element tv' IN Time_value,
              tv' <> tv  with  |T(tv) - T(tv')| <= epsilon
```

Second, consider that the waveform is made of points, as illustrated in figure 4(a). Each point represents an event with a specific value according to the waveform formula.

```
ENTITY point SUBTYPE OF (Event);
  point_value: REAL;
-- the instantaneous waveform value
END_ENTITY;

ENTITY waveform ABSTRACT SUPERTYPE; -- SIN(period * t + phase)
  epsilon, period, phase: REAL;     -- Variant 1
  points: SET [0 : ?] OF point;

WHERE
  epsilon_constraint: epsilon > 0;

  epsilon_time_continuous: -- definition 3
  QUERY(p1 <* points |
           (QUERY(p2 <* points | is_after(p1,p2)) = []) OR
           (QUERY(p2 <* points |
                      is_after(p1,p2) AND
                      (ABS(T(p1.event_time)-T(p2.event_time)) <= epsilon))
            <> []))
  = points;

  formula_constraint:
  QUERY(p <* points | p.point_value =
                      SIN(period * T(p.event_time) + phase)) = points;
END_ENTITY;
```

An additional ordering constraint can easily be added to make sure that `is_next(points[i], points[i-1])` holds for `i > 0`. However, this constraint is not essential, since the `Event` set is already ordered according to the `is_next` predicate, as proved in section 5.

Since `Time_value` satisfies property (a), and due to the `epsilon_time_continuous` constraint, it follows that the `points` set satisfies definition 3. Therefore, the modelled behaviour is epsilon time-continuous, for an appropriate value of epsilon. However, since epsilon is generic, the model can be taken as a generic model of the time-continuous behaviour of a SIN waveform.

The waveform model above is not the only possible alternative. Another variant, which does not incorporate epsilon and captures indirectly the condition from definition 3, is given below.

```
ENTITY waveform ABSTRACT SUPERTYPE; -- SIN(period * t + phase)
  period, phase: REAL;              -- Variant 2
  points: SET [0 : ?] OF point;

WHERE
  formula_constraint:
  QUERY(p <* points | p.point_value =
                      SIN(period * T(p.event_time) + phase)) = points;
END_ENTITY;
```

```
RULE compact_waveform FOR (Time_value, waveform);
  -- for any w IN waveform and t IN Time_value,
  --     there exists p IN w.points such that p.event_time = t
WHERE
QUERY(t <* Time_value | (is_greater(t,time_start) OR
                         is_equal(t,time_start)) AND
        (QUERY(w <* waveform |
                SIZEOF(QUERY(p <* w.points | is_equal(p.event_time,t))) = 1)
         = waveform))
= Time_value;
END_RULE;
```

The time model used in the waveform example is considered measurable. Nevertheless, the time values are not essential to modelling time. In addition, the `Time_point` and `Event` sets are generic and also the `past` and `present` event sets of a time point are unbounded, although finite. It follows that the basic time model is capable of representing what may turn out to be continuous time-based behaviour.

The time-continuous model of the SIN waveform can be seen as the limit of epsilon time-continuous behaviour models when epsilon value decreases. Ideal objects of this kind abound in mathematics, where an explicit notation is used for specifying them. It can be argued that, from the notational point of view, "limit" models should also be explicitly marked. However, it is debatable whether such extensions are really necessary for the modelling task solely and if the ABSTRACT SUPERTYPE specification facility does not suffice.

8.2 Modelling time-space phenomena

Section 8.1 illustrated the model of a particular time-continuous phenomenon. However, it is possible to create more general models to cover classes of phenomena which evolve in the 4-dimensional space: time and geometrical space. As an example, consider modelling the behaviour of objects which move through space and are characterised by the value of a single property (eg. light intensity they emit). Let call "pulsar" such an object. Therefore, the instant status of a pulsar is fully specified by the value of its property as a function of its position along the path it follows in the 4-dimensional space

```
position: (x,y,z)= (fx(t,y,z),fy(t,x,z),fz(t,x,y))
property: value= fv(t,x,y,z)
```

Considering that the linear and measurable time model is used, and that the `Time_value` is as in section 8.1, the generic pulsar can be described by the model below.

```
ENTITY time_quanta;
  epsilon: REAL;
WHERE valid_epsilon: epsilon > 0;
END_ENTITY;
```

```
RULE unique_time_quanta FOR(time_quanta);
WHERE
  SIZEOF(time_quanta) = 1;
END_RULE;

ENTITY point SUBTYPE OF(Event);
  point_value: REAL;  -- pulsar property
  x,y,z: REAL;        -- position in space

-- a pulsar is characterised by a set of points
-- in the 4-dimensional space (time * X * Y * Z)
END_ENTITY;

RULE dense_points FOR(point,time_quanta);
-- any point which has at least one predecessor is epsilon
-- close in time from at least one of its predecessors

WHERE
  QUERY(q <* time_quanta |
  QUERY(p1 <* point |
            (QUERY(p2 <* point | is_after(p1,p2)) = []) OR
            (QUERY(p2 <* point |
                      is_after(p1,p2) AND
                      ABS(T(p1.event_time)-T(p2.event_time))
                      <= q.epsilon)
             <> []))
  = point)
  = time_quanta;
END_RULE;

ENTITY epsilon_time_continuous_pulsar ABSTRACT SUPERTYPE;
  points: SET [1 : ?] OF point;
  begin_time, end_time: Time_value;

WHERE
  valid_begin_time: is_greater(begin_time, time_start) OR
                    is_equal(begin_time, time_start);

  valid_end_time:   is_greater(end_time, begin_time) OR
                    is_equal(end_time, begin_time);
  valid_point_time:
  QUERY(p <* points | (is_greater(p.event_time,begin_time) OR
                       is_equal(p.event_time,begin_time)) AND
                      (is_greater(end_time,p.event_time)   OR
                       is_equal(end_time,p.event_time)))
  = points;
END_ENTITY;
```

A particular pulsar can be, for instance, a source of light such that `value= SIN(period * t + phase)` and `x= y= z= value`.

```
ENTITY sin_pulsar ABSTRACT SUPERTYPE
       SUBTYPE OF (epsilon_time_continuous_pulsar);
-- particular type of pulsar
-- position unspecified, value= SIN(period * t + phase)
   period, phase: REAL;

WHERE
  sin_formula_constraint:
  QUERY(p <* points | p.point_value =
                      SIN(period * T(p.event_time) + phase)) = points;
END_ENTITY;

ENTITY static_sin_pulsar SUBTYPE OF (sin_pulsar);
-- position fixed, value= SIN(period * t + phase)

WHERE
  static_pulsar:
  QUERY(p1 <* points |
           QUERY(p2 <* points | (p1.x = p2.x) AND
                                (p1.y = p2.y) AND
                                (p1.z = p2.z))
           = points)
  = points;
END_ENTITY;

ENTITY sin_propagating_sin_pulsar SUBTYPE OF (sin_pulsar);
-- SIN moving SIN pulsar
-- x= y= z= value= SIN(period * t + phase)

WHERE
  changing_position:
  QUERY(p <* points | (p.x = p.point_value) AND
                      (p.y = p.point_value) AND
                      (p.z = p.point_value))
  = points;
END_ENTITY;
```

Assumming that geometrical space can be modelled by using numbers and sets [SG93], as time has been modelled, the example above proves that complex time and space dependent phenomena can actually be modelled in EXPRESS by using only basic concepts and that the semantics of such phenomena can be captured by structure solely. The evolution of such a phenomenon, in space, time, or both, is characterised by a set of points which fulfills specific content and structural constraints.

9 Conclusions

It is debatable whether or not the time models are really general. In any case the use of the word "general" is inappropriate here. Without knowing the exact nature of time it is difficult to claim that a model of time is general. Therefore, the word "sensible" is more suitable to underline the fact that the model describes time in a way close to our common sense perception of time. In addition, the models developed are intended for restricted use, mainly in modelling engineering artefacts, the behaviour of which is time dependent.

Nevertheless, the time models in this report and the simple example from section 7 show that time relationships can be reduced to structural relationships and specified in the same way as may be done for geometric, taxonomic and other "spatial" relationships. The model proves that time values are not necessarily important to express time-dependent behaviour.

Besides the chronological ordering of events, the chronological ordering of objects can be controlled provided that these are associated with events. Durations can be expressed as distances between events (start, stop) and, if necessary, measured by differences between time values. Moreover, it is interesting to note that time models are somewhat similar to the model of the electromagnetic field in modern physics. They are both corpuscular - by force of events - and continuous - by force of structure which provides for infinitely "close" events.

The possibility to model time (and space as shown in [SG93]) by using only two basic concepts, NUMBER and SET, and appropriate mechanisms for prototyping and constraint specification, suggests that the semantics of complex phenomena can be modelled in the same way, solely by using structure.

The models of time described here have already been used to model the behavioural aspects of VHDL by using the EXPRESS language [GC92]. However, their applicability extends beyond the domain of VHDL or even CAD. All time models given in the report have been checked by using the DEC EXPRESS toolkit (DEC & CADDTEC, 1992).

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 Hilary J. Kahn, Alan R. Williams, R.A.J. Marshall (Manchester University), Gerhard Scholz (Fernuniversitat Hagen), and Wolfgang Mueller (Paderborn University) for their useful comments and suggestions on the models.

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

[RF88] Flood R. and Lockwood M. (ed.) The Nature of Time. Basil Blackwell Ltd., 1986, 1988

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

[GC92] Giumale C. A. Towards a VHDL Model. Report ECIP2/UM/008-1, University of Manchester, 1992

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

[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

[SG93] Scholz G. A Conceptual Model for Geometry in Electronic Design (First Outline, January 11th, 1993). Unpublished Paper, University of Hagen

[VT92] Van Tassel J. P. A Formalistation Of The VHDL Simulation Cycle. Technical Report No.249, Computer Laboratory, University of Cambridge, 1992

[WE78] Weinberg S. The First Three Minutes. Fontana, London, 1978

[WD92] Wilsey P. A. and Davis K. A. Modeling VHDL's Execution Semantics in EXPRESS. Unpublished paper, 1992