The Hardware Description Language DACAPO III

Franz J. Rammig

Universität-GH-Paderborn
FB Mathematik/Informatik
D - 4790 Paderborn
Tel.: 49.5251.60.2069
Fax.: 49.5251.60.3427
e-mail: franz@uni-paderborn.de

Abstract

The broadband HDL DACAPO III is introduced by discussing language support for various levels of abstraction: Gate/Switch level, RT level, Algorithmic level, System level. A short comparison with the VHDL approach to model hardware by concurrently active sequential processes concludes this contribution.

1. The Hardware Description Language DACAPO III

DACAPO III is the most recent version of a hardware description language the first version of which was presented by the author in 1975 [Ra75], called DIGITEST II in those days. In 1979 a version very similar to the present one was defined and implemented. This language was called CAP/DSDL for Concurrent Algorithmic Programming Language/Digital Systems Description Language [Ra80]. With a different implementation the same language is called DACAPO II. Slight modifications and the addition of the MODULA II - like module concept resulted in DACAPO III [DA87].

In contrast to VHDL, DACAPO III is a real broadband language with strong support for the System Level, the Algorithmic Level, the Register Transfer Level, and the Gate Level. Weak support is provided for the Switch Level. In this contribution an informal introduction to the basic principles of the language will be given. It is organized with respect to the levels of abstraction to be covered. At the beginning however some common foundations will be discussed.

2. DACAPO III Basics

DACAPO III is a language that looks like MODULA II [Wi82] (or PASCAL) as far as possible. Therefore the basic notations, constants, identifiers, scope of variables, data types can be explained very briefly.

There is a relatively large number of reserved keywords and predefined identifiers that may not be used as user defined identifiers. In this description keywords will be marked by boldface.

Constants :

Numerical constants may be given in decimal notation or in a generalized binary one. *Decimal constants* are given by an arbitrary sequence of decimal digits, potentially with a leading sign. The range is extremely large as the virtual DACAPO-engine

J. P. Mermet (ed.), Fundamentals and Standards in Hardware Description Languages, 395–409.

is assumed to have a wordlength of `(2**31)-1` bits.

Bitstring constants are enclosed in leading and trailing ' " '. For each bit-position there exists the extended range {`0, 1, X, L, H, Y, Z`} with the following meaning:

0	:	logical zero	,	low impedance
1	:	logical one	,	low impedance
X	:	logical uncertain	,	low impedance
L	:	logical zero	,	high impedance
H	:	logical one	,	high impedance
Y	:	logical uncertain	,	high impedance
Z	:	no logical value	,	high impedance

Bitstrings are also interpreted as integers and vice versa. *Characterstring constants* are enclosed in ' ' ' symbols. Each enclosed character is interpreted as a bitstring of length 8 according to EBCDIC or ASCII code, dependent on the host machine's default code.

Data types :

The basic data type of DACAPO III is the bitstring of arbitrary length. It is denoted by **bit**(n) where n gives the length in bits. Each "bit" is seven valued as explained above. To express this explicitly one may also write **bit_7**(n). Instead of writing **bit**(1) or **bit_7**(1) one may simply write **bit** or **bit_7**. If one wants to restrict the "bits" to the range {0,1,X} one has to write **bit_3** instead of **bit** or **bit_7**. The type **integer** denotes a bitstring of length 32 where each "bit" is restricted to the domain {0,1}. Similarly by the type **timevar** such a string of length 64 is denoted. In any case the bits are counted from right to left, starting with bit number 0. While in bitstrings the actual coding is given, this is left open in *enumeration types* where the range is simply given by an enumeration of the range of possible values.

Examples :

bit
bit_3*(3948567093459846987654)*
(andcode, orcode, nandcode, notcode, norcode, exorcode, addcode, minuscode)

Structured types are made by the PASCAL-like constructs for *arrays* and *records*. *Arrays of records* and *arrays of arrays* are allowed. However arrays are not allowed as record-components. The reason is that a record is also looked at as the bitstring that results from concatenating all its components.

Examples :

array [1023 : 0, 0 : 7] **of bit**
this is equivalent to **array** *[1023 : 0]* **of array** *[0 : 7]* of bit
array *[0 : 255]* **of record**
opc : **bit***(3);*
adr : **bit***(13)*
end

Type definitions :

Data types are declared in a PASCAL-like fashion. In DACAPO the type-concept is generalized to include *Abstract Data Types* as well. This will be discussed later.

Object declaration :

Objects of certain (user-defined or predefined) types are created by declarations. Besides a generalization to allow instantiations of ADTs (will be explained later) this is done asin PASCAL. Data objects may get an *initial value* by assigning a constant (expression) to the declaration. The default initial value is **"ALL_Z"** DACAPO distinguishes between two main classes of data objects:

- objects *with storing capability* (registers, memory cells, Flipflops) and
- objects *without storing capabilities* (connection lines, outputs of combinational logic).

Storing objects are denoted by the attribute **explicit** (which can be omitted) while non storing objects have to be attributed **implicit.**

Expressions :

Expressions are similar to expressions in PASCAL. References to the current value of a simple data object is made by simply naming the object. A current value of an array-component-object is referenced by naming the array-object together with the proper index(-list). Entire arrays may be referenced as well, just by omitting the index(-list). In the case of multidimensional arrays only the rightmost index or the entire index-list may be omitted. The actual value of a record-object is referenced by naming the complete path to the component using a dot as delimiter. Entire records may be referenced as well as being treated identical to bitstrings.

There is a wide range of operators in DACAPO for arithmetic, logic, comparison, string-manipulation, and case distinction. Only the last one shall be discussed here.

Case distinction :

If E_1 and E_2 are valid expressions and *cond* is a data object of type **bit**(1) then **if** *cond* **then** E_1 **else** E_2 is a valid expression. The value of the entire expression is either the value of E_1 or of E_2, dependent on the actual value of *cond*. Note that obviously the *else part* must not be omitted.

If $E_0, E_1, ..., E_n$ are valid expressions, *cond* is a data object and $\{v_0, v_1, ..., v_{n-1}\}$ is a subset of the range of *cond*. Then **case** *cond* **of** $v_0 : E_0$; $v_1 : E_1$; ...; $v_{n-1} : E_{n-1}$; **else**
E_n **end** is a valid expression. The value of the entire expression is the value of E_i if the current value of *cond* is v_i.

Function references :

Instead of referencing a data object a *function* may be referenced. The referenced function may either be a language provided *built-in function* or a user defined one. A function may have arguments (*formal parameters*) that have to be served by *actual parameters*. Again strong type checking takes place. Definition and use of functions will be discussed in more detail when discussing DACAPO descriptions at the System Level (section 4)

Assertions:

Simulation of a system is an attempt to verify it. This means one tries to check out whether the described behavior matches the intended one. So the designer knows the conditions that have to be true. Usually he has to analyze the simulation result for checking the match. However, if he is able to formulate the conditions in advance, he can hand over this cumbersome task to the simulator. In DACAPO he has the option to formulate his conditions as *assertions*. Such assertions are invariants that have to hold all time. In case of a violation the simulator reacts in a user definable way, e.g. by printing an error message. All assertions of a procedure or function are grouped together in an assertions-part, headed by the keyword *assertions*. Each assertion is of the form:

condition → *action*

Here *condition* is an arbitrary expression of type **bit***(1)* and *action* is an arbitrary statement. The *assertion* is evaluated continuously (the simulator makes use of special techniques to do this with a minimal amount of cpu-time). Whenever *condition* becomes true, *action* is executed. Typical actions are :

- Modification of the state using an assignment statement,
- error-message using built-in function **error**,
- stop of simulation using builtin function **stop**.

3. Descriptions at the Algorithmic Level

Algorithms play an important role at the System Level as well but they are centered at the Algorithmic Level. Therefore this level is discussed first. Similarly all language support for modularization will be discussed in the section about the System Level, though these techniques are of importance at the other levels as well. The algorithmic part of a DACAPO description consists of a *compound statement*. Such a *compound statement* (CS) consists of a CS- head, a list of statements and a CS-end which is simply the keyword **end**. The following types of statements may occur within a compound statement :

- *compound statement*,
  assignment statement,
  if-statement,
  case-statement,
  while-statement,
  repeat-statement,
  for statement,
  at/when-statement,
  procedure-call,
  empty statement.

3.1. Compound statement

There are four different types of *compound statements* :

- The *sequential compound statement*,
- the *concurrent compound statement*,

- the *parallel compound statement*,
- the *compact sequential compound* statement.

The last one will be discussed in section 7 (behavioral descriptions).

The *sequential compound* statement has the form

seqbegin S_1; S_2; ...; S_n **end** ;

The semantics is that statement S_i is initiated after statement S_{i-1} has been terminated. However this does not necessarily mean that S_i is initiated immediately after statement S_{i-1} has terminated as there may be concurrently active parts that may interfere with this statement.

The *concurrent compound statement* has the form

conbegin S_1; S_2; ...; S_n **end** ;

The semantics is that by initiating the entire *concurrent compound statement* all embedded statements S_1; S_2; ...; S_n are initiated concurrently. They are now executed completely independently. When the last one of these statements has terminated the entire *concurrent compound statement* terminates. It should be noted that the ordering of the statements S_1; S_2; ...; S_n within a *concurrent compound statement* has no semantical meaning.
The following example shows why two consecutive statements in a sequential compound statement are not necessarily executed immediately one after the other:

```
conbegin
  seqbegin
    a := 1 ;
    b := 10/a
  end ;
  a := 0
end
```

In this example a *concurrent compound statement* contains two statements: A *sequential compound statement* and a simple *assignment statement*. As there is no inner synchronization between the two statements of the *concurrent compound statement* it may happen that the single *assignment statement* a := 0 is executed just after the first *assignment statement* of the *sequential compound statement* (a := 1) but before the second one (b := 10/a), potentially causing some unintended problems.

The *parallel compound statement* has the form

parbegin S_1; S_2; ...; S_n **end** ;

In this case the embedded statements S_1; S_2; ...; S_n are restricted to *assignment statements*. The semantics is that all these statements are initiated when the entire *parallel compound statement* is initiated. They are executed in a strictly synchronized manner. That is the execution of a statement S_i does not affect the other statements of the same *parallel compound statement* at all. Of course the ordering of the statements S_1; S_2; ...; S_n within the parallel compound statement has no semantical meaning.

Example :

parbegin a := b ; b := a **end** ;

This describes a simple swap operation while

conbegin a := b ; b := a **end** ;

would describe a nondeterministic behavior.

3.2. Assignment statement

An assignment statement causes a new value at a storing data object. The receiving data object will keep this value until another assignment takes place. The assignment target may also be formed by a concatenation of data objects, a substring, or a combination of both. In addition multiple assignment is allowed. It assigns an identical value to a list of data objects.
Assignments may be delayed to describe realtime behavior.

3.3. If - Statement and Case-Statement

The *If-statement* enables alternative flows of execution to be described. The decision is made according to the actual value of a data object or expression of type **bit**(1).

3.4. Case - Statement

Like the *If-statement* the *Case-statement* describes alternative flows of execution. In this case, however, one is not restricted to two alternatives but can select between an arbitrary number of branches. The decision is made according to the actual value of a data object or expression.

3.5. While - Statement and Repeat-Statement

The *While-statement* is the basic DACAPO form of a loop.
The *Repeat-statement* is very similar to the *While-statement*. The main difference is that the loop-body is executed first and then the test takes place.

3.6. For-Statement

Syntactically the DACAPO *For-statement* is similar to the For-statement of PASCAL. However, a different semantics is attached in many cases. The *For-statement* has the following form:

for *index* := *start_value application_selection final_value* **by** *step_size*
do *S* ;

Here *S* is an arbitrary statement (see restrictions below), *index* denotes a data object while *start_value*, *final_value* and *step_size* are constants (or constant expressions). The term *application_election* has one of the following forms :

- **seqto, seqdownto** {sequential application},
- **conto, condownto** {concurrent application},
- **parto, pardownto** {synchronized parallel application},
- **to, downto** { compact sequential application}

Semantics: In DACAPO the For-statement does not denote a loop. It is just shorthand for a compound statement. Let be # ε {**seq, con, par, ,**}
and let $\mathbf{S}_{index \rightarrow value}$ denote statement **S** with each occurrence of index replaced by value. Then

for *index* := *start_value* # **to** *final_value* **by** *step_size* **do** *S* ;
is equivalent to:

```
# begin
    S_index→start_value;
    S_index→start_value+step_size;
    S_index→start_value+2*step_size;
    .
    .
    .
    S_index→final_value
end;
```

3.7. At/When Statement

Usually in imperative programming the flow of control is given entirely by the control structure of the algorithm. This complicates it a little bit to describe synchronization of the algorithm with external events. Such events may ocur as single ones (e.g. a keystroke on a keyboard) or continuously (e.g. a clock). To support this, in DACAPO a synchronization structure may be overlayed on top of a given control structure. The basic idea is that any statement may be attached to an event it has to synchronize with. In such a case the statement is executed if it has to be executed according to the "normal" control structure and the synchronization event occurs. The synchronization event may be either a certain value transition or a level. The *At-statement* describes edge-triggered synchronization. It has the form :

at *direction (event)* **do** *statement* ;

Statement is an arbitrary statement. This statement will be executed if it is initiated according to the "normal" control structure and after this the event has become true.

The When-Statement describes the level sensitive alternative. It has the form:

when *condition* **do** *statement* ;

3.8. Procedure call

Organizing algorithms with the aid of procedures is an important structuring method. As in this context the entire modularization mechanism of DACAPO is described in section 4 (Descriptions in DACAPO III at the System Level) a detailed description of procedure calls is delayed to this section.

Activating a procedure call does not necessarily result in an immediate initiation of its activity to serve this request as there may be competing concurrent requests to a single procedure object that can serve only one request a time. See section 4 for more detail.

3.9. Empty statement

The *empty statement* is syntactically given by an empty string as well.

```
seqbegin
    if a <> b then c := a delay ( loadtime )
    else delay ( prechargetime ) ;
    delay ( loadtime ) ;
end ;
```

In this example two *Empty-statements* are included. The first one is used to "simulate" an absent *else* branch and is used to describe that some time (which is the actual value of data object *prechargetime*) has to elapse. The second *Empty-statement* is used only to describe that some time has to elapse after the termination of the

If-statement until the entire *sequential compound statement* terminates. Note that unlike VHDL in DACAPO, it is assumed, that a statement with delay consumes time, i.e. the initiation of the following statement is delayed.

4. Descriptions at the System Level

In this section a couple of language concepts will be described that are of value at other levels of abstractions as well. As the main support of System Level descriptions originates from modularization and encapsulation all related language features are discussed here. Also high level event driven descriptions will be described in this section. A DACAPO description is composed of *Modules.* Here there is made the distinction between *Definition Modules* that specify the interface to the environment and *Implementation Modules* that specify the internals. The outermost module is just a *Module* without separation into Definition Module and Implementation Module. Modules may be further organized using Procedures, Export Procedures, and Functions.

4.1. Procedures

In DACAPO *procedures* are the basic technique to block-structure a description.

A *procedure* is activated by calling its identifier and binding actual parameters to formal ones. Each actual parameter has to be Type-compatible with its corresponding formal one. Parameters of type **explicit** are passed with their value at the time of calling and receive the value of the respective formal parameter at time of termination of the called *procedure.* Parameters of type **implicit** are kept at equal value continuously.

A *procedure*, once activated, remains active until its *algorithmic part* becomes terminated. In contrast to languages like PASCAL all variables of type **explicit** keep their value when the *procedure* is deactivated. The rules concerning the scope of identifiers is exactly that of PASCAL. An important feature of a *procedure* is that it can be active only once at a certain point of time. On the other hand there may be concurrent attempts to activate it. In such a case a builtin arbitration mechanism selects the request to be served first.

4.2. Functions

Functions differ from procedures only in that they return a value bound to the object identified by the function identifier. Like *procedures, functions*, too, can be active only once at a certain time. So they too have a builtin arbitration mechanism to resolve conflicts due to concurrent activations.

4.3. Export Procedures

Export procedures are the DACAPO notation for *Implemented Abstract Data Types* (IADT) [GH78], if it is required that all operations of the IADT have to be mutually exclusive. If this restriction is not required or not desirable, IADTs can be specified with *modules* as well (see section 4.6). An *export procedure* is given by an *export procedure head* and an *export procedure body.* The *export procedure head* has the following form :

export *(list of operations)* **procedure** *export_procedure_identifier ;*

The body of an *export procedure* is very similar to the body of an ordinary one. However, the *algorithmic part* has to be replaced by the keyword **end**. For each identifier contained in the list of operations there has to exist exactly one *function* or *procedure* with the same identifier that defines the implementation of this operation. The operations are mutually exclusive simply by the fact that an *export procedure* is a *procedure*. By this it can be active only once at a certain point of time.

4.4. Types of Procedures, Functions and Export Procedures

In DACAPO the PASCAL type-concept is generalized in such a manner that types of *Procedures, Functions* and *Export Procedures* are allowed as well. This is done by simply using a *Procedure-/ Function-/*, or *Export Procedure-declaration* as replacement of the *type description* within a *type definition.* Instances of objects of such types are made just by declaring variables of such a type.

4.5. Generic Types

In many cases it is desirable to have instances of a certain type that differ slightly from each other. For example it might be desirable to have available a fifo-buffer where instances of various depths and wordlengths are possible. This possibility is provided by *generic types.* To make a type generic, the type description has to be prefixed by a *generic specification.* This has the form :

generic *list-of-generic-attributes :*

If an object of this type is instantiated, *actual attributes* have to be provided for such formal ones. This is simply done by postfixing a list of actual attributes (types or constants), separated by commas and bracketed by square brackets.

It should be noted that the generic concept is not restricted to *Procedures, Functions*, and *Export Procedures* but may be applied to any types.

4.6. Modules

A *Module* is a compilation unit in DACAPO III. So the module concept not only enables descriptions to be structured, but also libraries of (pre-) compiled object descriptions to be kept.
There are two main classes of *Modules* : *Definition Modules* that specify the interface of a *module* to the environment i.e. those internal objects it wants to make visible to the environment, and *Implementation Modules* that describe the interior of a module. For each *module identifier* there has to be exactly one pair of one *definition module* and one *implementation module.*
A *Definition Module* makes visible an internal object just by listing it by identifier and type. Note that the *formal parameters* of *Procedures* and *Functions* are part of their type-definition. If a *module* (*Definition Module* or *Implementation one*) wants to make use of an object offered by another *Definition Module* it has to import it. This is done by *import clauses* just after the module head.

The outermost (*Implementation-) Module* has no *Definition Module* attached.

4.7. Interrupt Systems

DACAPO offers an event driven modeling style at higher levels, too. This is done using an *interrupt concept.* Of course, *interrupts* in a highly concurrent environment

have a semantics which is not completely identical to that in sequential systems. The basic idea of an *interrupt*, however, to transfer a system into a specific state, independently from its present state is maintained in our context. The *interrupt concept* is established using *Interrupt Signals*, *Interrupt-Service* routines, and *Operations on Interrupt Signals*. *Interrupt signals* are objects of a specific type :

interrupt *(priority)*

An *interrupt signal* can have only two values:

{set, reset}.

Reacting on *interrupt signals* are *interrupt service routines*. If an *interrupt signal* gets a value *"set"* every currently active *procedure* or *function* with an *interrupt service routine* for this *interrupt signal* declared in it is interrupted. All other currently active *procedures* and *functions* remain unaffected. In an interrupted *procedure* or *function* first the *interrupt service* routine is executed, then the activity is resumed at the situation when the *interrupt* occurred.

It should be mentioned that by this concept one *interrupt signal* may be served by more than one *interrupt service routine* concurrently and that the actions within these *interrupt service routines* may differ. A system reset signal is an example for such a situation. It causes different modules to perform a local reset operation that may be very specific to the very module.

An *interrupt service* routine has the following meaning: If the *interrupt-signal* becomes set then the *procedure* or *function* it is declared in is interrupted, including all dynamically activated *procedures* or *functions*. However, this takes place only if this block is currently active. Interruption of an active *procedure* or *function* means that all currently active non interruptable actions (assignments, empty statements, compact compound statements) are processed as normal but no more actions are initiated.

When the last non interruptable action has been terminated the *interrupt service routine* is initiated and all *interrupt signals* that caused this interrupt are reset locally for this *interrupt service routine*. They may remain *set* at other ones. After the termination of the *interrupt service routine* the interrupted activity is resumed. i.e. all actions that would have been initiated in the case of no interruption are initiated now. Values of interrupt signals are stored for *procedures* or *functions* that are not active at the time when the interrupt signal was set. They get interrupted immediately upon activation. Interrupt signals can be manipulated and interrogated only with specific operations.

4.8. Protocol Specification

Besides the specification of the module's functionality, it has to be specified how it communicates with its environment. That is a *protocol* has to be specified. Protocols are described in DACAPO either by *interrupts* or using the *at/when* construct of the language. The *at/when-style* is more related to a *rendezvous*-like communication as the receiver in an arbitrary state cannot be forced to accept a message. Instead of this it has to become a message accepting state due to its flow of control. This is fine as long as the protocol assumes this.

5. Descriptions at the Register Transfer Level

At the Algorithmic Level systems are described in an imperative way. Even by introducing the *at/when*-construct there establishes only a subordinated level of additional synchronization within an imperative domain. But the imperative control structure may become meaningless if the control structure keeps all statements enabled continuously and concurrently. In such a case we have obtained a reactive description within an imperative domain. To ease the description in DACAPO such a reactive description is concentrated in a special part, headed by the keyword **impdef**.

Of course the order of the statements within an *impdef*-part has no influence on the semantics of such a description. By this we have obtained a Register Transfer language that is slightly more general than usual ones, as in most RT-languages the actions are restricted to assignments.
Assume that the actions are restricted to assignments with data objects of type **explicit** as target objects (storing variables, registers). Then by a statement of form:

at up *(event_1)* **do** *target := expression ;*

an edge-triggered transfer of a value into a register is described. This target register is sensitive on *rising edges* only. Registers triggered by *falling edges* are described by:

at down *(event_1)* **do** *target := expression ;*

If a *master-slave* operation is to be described the two guards just have to be combined:

at up *(event_1)* **do**
at down *(event_1)* **do** *target := expression ;*
or

at down *(event_1)* **do**
at up *(event_1)* **do** *target := expression ;*

If the buffer-register is of explicit interest this may be replaced by :

at up *(event)* **do** *master_target := expression ;*
at down *(event)* **do** *slave_target := master_target ;*
or

at down *(event)* **do** *master_target := expression ;*
at up *(event)* **do** *slave_target := master_target ;*

The values over the time axis of non storing data objects get defined in the **impdef**-part as well. For each such object declared there has to exist exactly one equation of the form :

target := expression ;

Though looking like an assignment in fact it is an equation in this case as the target value always has the value assigned, i.e the assignment is executed continuously (conceptually, in the case of simulation a more efficient alternative with equivalent result is used). Such an equation may be prefixed by **when** *condition* **do** resulting in a statement of form:

when *condition* **do** *target := expression ;*

Here *condition* is an arbitrary expression of type **bit**(1). The semantics of this statement is that target follows the value of *expression* as long as *condition* has value "1". If *condition* gets value "0" then target holds its most recent value until *condition* gets the value "1" again. So by this statement a *level-sensitive latch* with transparent mode during level "1" is described. As mentioned in section 3 each assignment-

statement may be delayed. Of course, this is true in this case as well, including the case where the assignment has become an equation in fact. It is described by postfixing a term **delay** *(delay-expression)*. This will be discussed in detail in section 6. Let us assume at the moment that *delay-expression* is just an expression of type **timevar** (i.e. **bit**(64), restricted on values "1" and "0" per bit). So a register transfer with delay may look like :

at up *(event)* **do** *target := expression* **delay** *(some_delay) ;*

The semantics is that the expression is evaluated immediately after event has become true based on the values of the data-objects involved at this point of time. On the same basis some delay is evaluated. The assignment of this value of expression however is delayed by some delay time units. Until this point of time target remains its old value. DACAPO knows no predefined implicit clock. So both, synchronous and asynchronous systems can be described. If a clock is needed this can be done simply by declaring a nonstoring data-object of type **bit**(1) and defining it in the *impdef-part* as its own complement with a proper delay.

6. Descriptions at the Gate/Switch Level

At the Gate Level a set of Boolean equations have to be provided. As already described in section 5 (DACAPO III Descriptions at the Register Transfer Level) this is done using **implicit** variables and assigning to them an expression within the **impdef** part. As the objects involved are not restricted to the type **bit**(1) bundles of functions can be specified in a concise manner. At least at this level the timing concept becomes very important. The default timing concept of DACAPO is *unit delay*. If no explicit timing information is provided it is assumed that each assignment takes exactly one time unit. By a global option the value of this unit can be set. There are no restrictions concerning this default delay value. So the value 0 is one of the options allowed. For a more precise timing the user is allowed to specify a *specific delay* for each assignment and for each empty statement. As the default delay is still assumed for all assignments where no explicit timing information is given it is advisable to set default delay to 0 as global option in this case. Explicit timing information is given by a postfix to assignment statements and empty ones. This postfix has the general form :

delay *(delay specification)* .

The general semantics is the following:
If an assignment is initiated, a snapshot of the current values of the arguments of the assignment and of the delay specification is taken. On the basis of these values the value E of the expression to be assigned and the value D of the delay specification are calculated. The assignment of E to the target variables of the assignment statement however is delayed until D time units have elapsed since the initiation of the statement. During this period the target variables keep the old value (if they are not affected by other assignments during this time). All value changes that may happen during this time to the arguments of the expression to be assigned and the delay specification have no effect on E and D. In the simplest case delay specification is just an expression, e.g. a constant:

delay *(35)*
delay *(gate_delay)*
delay *(latency_time + seek_time)*

The *delay-expression* may include a *case distinction* on the values of the expression to be assigned. Typically this is the case when gates are modeled more precisely. In this case usually the rise- and fall-times differ considerably.

a := b & c **delay** (**if** *b & c* **then** *rise_delay* **else** *fall_delay*) ;

This is a valid delay specification. However it is not only cumbersome to write down but also to calculate, as the expression *b & c* has to be calculated twice. Therefore DACAPO offers a shorthand for this situation which in addition is more efficient in runtime :

delay (**up** *delay_specification* , **down** *delay_specification*)

This delay specification may be used with all types of assignments. If the assignment target is not of type **bit**(1) the different bits of the target get their new value independently with the specified delay.

Example:
(Assume *a* is of type **bit**(2) and the statement is initiated at point of time t_o. Assume *a* had value "10" before)

a := "01" **delay**(**up** *10,* **down** *20)* ;

This results in the following sequence of values of *a* :

t_{o+10} : *a* = "11"
t_{0+20} : *a* = "01"

In many cases the exact delay time is not known but only a certain bandwidth. In order to describe this situation in DACAPO each delay specification may be of the form :

min_delay_specification **to** *max_delay_specification*

where *min_delay_specification* and *max_delay_specification* are arbitrary expressions.

Example:

delay (**up** *30* **to** *32,* **down** *22* **to** *38)*

This is interpreted in such a way that first an uncertain value is assigned and after the interval of uncertainty has elapsed the final definite value. So in the above example if there would be a value *"0"* to be assigned at t_0 then at t_{0+22} a value *"x"* would be assigned and at t_{0+38} a *"0"*, finally.

Gate Level descriptions may be given in *netlist-form* in the extreme case. In this case the expressions to be assigned are restricted to contain only one operator. This form of description documents very precisely the implementation structure by single gates. In the other extreme an entire combinational circuit with n primary outputs and m primary inputs may be described just by n expressions with up to m arguments each. A widely used compromise is to identify common subexpressions and assign them to intermediate variables (internal fan-outs) used as arguments by other expressions.

Switch Level Descriptions

The Switch Level is not supported by special language constructs in DACAPO. The 7-valued logic and a couple of builtin procedures together with the general power of the language, however, allow fairly precise switch level descriptions.

7. Behavioral Descriptions

The term *"Behavioral Language"* is very misleading. It is used for languages that describe the I/O function of an object only. In most cases algorithmic languages are used for this purpose despite the fact that a function has to be described. It is assumed that this function has to be calculated whenever a signal change at any primary input of the object occurs. The arguments of the function are the object's current state and the current values at its primary inputs. The function calculates the object's new state and the values at its primary outputs. All internals of the object are of no interest. This is exactly the wayÄlt gate models are handled in event driven gate level simulators. The so called "Behavioral Languages" originated from the intent to offer more complex "gates" to those simulators. VHDL processes originate from the same idea. Following a completely data driven approach, such models can not be used to specify or document concurrent control structures (algorithms). On the other hand this approach results in relatively fast simulation times as the object-models can be compiled into directly executable code. Although not intended for this purpose DACAPO can be used as behavioral language as well. For this purpose there is the special form of the *compound statement:*

begin ... end

This statement models a timeless non interruptable activity. Therefore the statements within such a statement are restricted. There is no *delay* and no *at/when* allowed and contained statements also are restricted in this way. Consequently only procedures and functions can be called which have such a compound statement as algorithmic part. With these restrictions a DACAPO style is obtained that is very similar to VHDL processes. So the following example is just a DACAPO version of a VHDL process that is sensitive on signals *arg1* and *arg2* and calculates signal *res* (multiplication)

```
at change (arg1 | arg2) do
  seqbegin
   begin
    intres := 0 ;
    for i := 0 to 15 do
     begin
      if arg1.(0) = "1" then intres := intres | + | arg2
         else ;
      intres || arg1 := shr(intres || arg1, 1)
     end
   end ;
   res := intres delay (30 + 30 * onecount(arg1) )
  end
```

Remarks: The **begin ... end** is timeless. So the timing behavior is described by the surrounding **seqbegin ... end**, or more precisely by the delayed assignment statement to be executed after termination of the **begin ... end**. The function *onecount* is assumed to be a user supplied function that returns the number of ones of an argument. Note that this function describes no hardware component but is just an auxiliary function to describe a certain feature of a hardware component. This is the other purpose of the **begin ... end** statement. So a synthesis algorithm can be biased to ignore such statements entirely.

8. Conclusions

DACAPO III as a result of a relatively long evolution is a real broadband HDL. It offers powerful support for the gate/switch level, the register transfer level, the algorithmic level up to the system level. Inheriting the module concept from MODULA 2 and providing easy to understand ADT notation, highly complex systems can be specified in DACAPO III. A very flexible and powerful delay model is offered and a formal Petri Net [Re85] based semantics which abstracts completely from a simulator is available. DACAPO III (and derivates) have been used successfully in industry for simulation and synthesis. Currently it is influencing the discussions on how to enhance VHDL, especially for system level support.

References

[DA87] -: DACAPO III - System User Manual. Dosis GmbH, Dortmund, 1987

[GH78] J. Guttag, J.J. Horning: The Algebraic Specification of Abstract Data Types. Acta Information, 10, 1978

[Ra75] F.J. Rammig: DIGITEST II: An Integrated Structural and Behavioural Language. Proc. IFIP CHDL'75, 1975

[Ra80] F.J. Rammig: Preliminary CAP/DSDL Language Reference Manual. Univ. Dortmund, Abt. Informatik, TR No. 129, 1980

[Re85] W. Reisig: Petri Nets: An Introduction. Springer, 1985

[Wi82] N. Wirth: Programming in Modula 2, Springer, 1982