Generic Design: Its Importance, Implementation and Limitations[1]

K. ten Hagen, H. Meyr

Aachen University of Technology, IS^2, Templergraben 55, D-5100 Aachen, Germany
phone: ++49-(241)-807632, fax: ++49-(241)-807631
email:{tenHagen, Meyr}@ert.RWTH-Aachen.de

Abstract

In traditional bottom-up design complex units are built up in a recursive process from simpler ones, starting with gates of a library. Therefore during the whole design process the speed and area of a unit are precisely known. This is not the case in top-down design, where many realisation parameters are fixed late in the design process. Therefore the efficiency of top-down design is increased significantly by generic design, where each entity is parameterised persistently. In a design project with three different ASICs, VHDL modeling guidelines for generic design have been established. These modeling guidelines are discussed using a subdesign from the design project. An organisation scheme for the design data supporting generic design is described. The limitations of such a generic design at different levels of abstraction are indicated. The implications of generic design to design reusability and future system development environments are discussed as well.

1 Introduction

In traditional bottom-up design complex units are recursively built up from simpler ones. The process starts with gates from a library supplied by a silicon vendor. Therefore the level of abstraction applied throughout the whole design process is the gate level [1]. This approach has two main disadvantages: (1) simulations to study the initial ideas are available too late in the design process and (2) the runtime of these simulations is excessively long. Nevertheless after a unit has been designed, the clock frequency and the silicon area occupied are precisely determined.

On the other hand top-down design starts with an abstract model on a higher level of abstraction, e.g. the functional level [2]. This model is successively decomposed and stepwise refined [3]. Many realisation parameters, like the range of data items, are determined by simulations at different levels of abstraction, some not even until placement and routing (P&R) of the cells on the chip. Therefore top-down design has to cope with a large amount of uncertainty during the whole design process.

It was one of the major lessons learned during a complex design project, which comprises approximately 40.000 lines of VHDL to model three different ASICs, that this uncertainty can only be effectively managed by *generic design*. Goal of the project was to speed-up the runtime behaviour of rulebased expert systems by an add-on board, called *AcE* (Accelerator for Expert Systems) to allow an application within a real time environment [4].

Generic design consists of:

- Analysis of the *realisation parameters*.
  - Determination of the *independent realisation parameters*.
  - Derivation of each *dependent realisation parameter* from the independent ones.

  The independent parameters may in principle be arbitrarily fixed by the designer.
- Making the necessary realisation parameters accessible for each model.
- Modeling each entity for the complete range of its realisation parameters, as far as this is possible at the actual level of abstraction.

[1] This work was supported by ESPRIT 2434 and the EuroChip-Project.

- Integration of models optimised for a particular set of realisation parameters without affecting generality.
- Organising the design data, making the dependencies managable and minimizing the recompilation effort after change of a parameter.

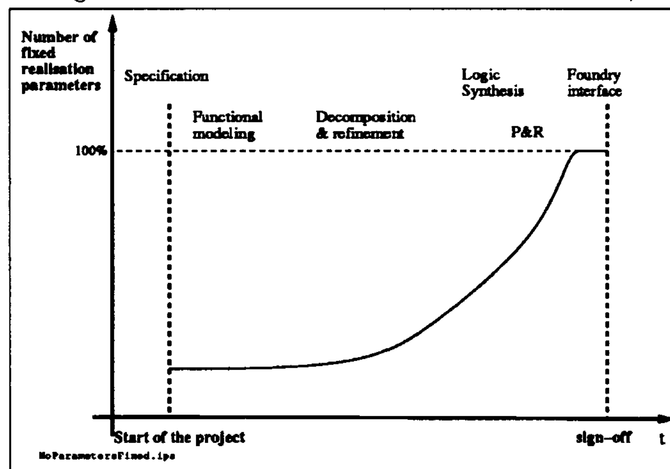
In the following section the importance of generic design within a top-down design project will be discussed. In section 3 the implementation of generic design at different levels of abstraction is described and various guidelines for generic design are developed. A small case study from the design project will be used to demonstrate the implementation of generic design. In section 4 the limitations of generic design will be outlined. In the last section 5 the impact of generic design to reusability of verified models and the influence of generic design on future *system development environments* will be given.

2 Importance

At the beginning of the project in 1989 [3] we did not follow the above mentioned guidelines and thus quite frequently VHDL models, considered as stable, needed to be amended after a single realisation parameter had been changed.

2.1 Redebugging after a Parameter Change

The number of realisation parameters increases during the design process, since many realisation parameters are introduced during structural decomposition. Furthermore, realisation parameters are fixed through simulations at different levels of abstraction, some not even until P&R. Therefore the number of realisation parameters fixed within a specific period increases during the design project, as the understanding of the system increases steadily. Furthermore, quite often the necessity occurs that some parameters are to be changed in the *last minute* before *sign-off* [5]. The number of fixed realisation parameters is sketched over time in the adjacent figure. Debugging in the final stage tends to be a critical task, since the models are the most complex and deadline pressure is high. The reasons for such failures were either a lack of transfer of the realisation parameter through all levels of hierarchy to the unit or the definition of a parameter as a constant within the unit itself, since the parameter was not identified by the designer of the unit as a realisation parameter.



2.2 Arbitrary Fixing of a Parameter

Bottom-up design of a unit can only start after all realisation parameters of that unit have been fixed. Therefore designers now involved in top-down design tend to fix realisation parameters too early and without necessity. Such an arbitrarily fixing can create various problems:

1. Simulations to analyse the real requirements for this parameter are not made, since nobody has been made conscious of this hidden parameter.
2. Hardware resources, like registers or interfaces, are oversized on the real chip.
3. The whole system simulates nicely until a parameter change, not compatible with the fixing, is executed.
4. Dispensible performance losses occur, since units, like queues, are dimensioned too small.

At the first glance items 2 and 4 seem to be in contrast to each other. However 2 occurs, since realisation parameters of an entity are fixed for the worst case imagined by the responsible designer, and 4 does happen, since the influence of the dimensions of the actual unit on the performance of the whole system is not understood at the time the realisation parameters are unnecessarily fixed.

The above dicussed problems can only be avoided if every designer is aware of the fact that top-down design without following the guidelines of generic design can lead to severe problems in the final stages. Therefore the first guideline of generic design is:

Guideline 1 (Awareness) *Make every designer involved in the project aware of the principles of generic design.*

In order to avoid that realisation parameters are arbitrarily fixed by the designer of the unit and hidden somewhere in the design data, the following guideline should be obeyed:

Guideline 2 (Documentation of Realisation Parameters) *The documentation of each entity should comprise a list of its realisations parameters.*

3 Implementation

Generic design is indispensible for top-down design and can be used to improve reusability in bottom-up design using VHDL[2] as well. However in the following generic design will be discussed within the context of top-down design.

Top-down design is the recursive replacement of an abstract behavioural model of an entity by a structural model instantiating entities with a less abstract behavioural model. This process comes to an end, if every behavioural model can either be effectively synthesized by a logic synthesis tool or merely instantiates gates/macros from a silicon vendors library. [2]

Therefore top-down design starts with the creation of an abstract model of the specifications, which can be simulated.

3.1 Functional Model

The first model used to understand the specification and to study the behaviour of the applied algorithms will be implemented in the most appropriate language [6]. This can be a domain specific language, like *SDL* [7], or a domain specific system development environment like *COSSAP* [8]. In the considered design project the first functional model was implemented in *C*, since the runtime efficiency was much better than for VHDL [2] and tools, like a runtime profiler [9], are not available for VHDL. Various realisation parameters have been determined with the functional model in C, since complex benchmarks can only be completely executed with this model.

3.1.1 Case Study

The functionality of the subdesign used as a case study is to store objects. The objects are created and removed dynamically. Each object consists of a number of so-called atoms. Single atoms of an object are read. The value of an atom is either an integer or an array of characters (string). The strings are encoded by a hash index [10] as they are entered into the system at the host computer to which the AcE board has been added. Therefore an atom is modeled at this stage in the standard definition file *stddef_p.vhdl* as shown in the left code fragment. This definition neglects the fact, that on a real chip the value of an atom must be constrained to fit into allocated hardware resources, like registers or data paths. However it is good modeling practice to define the data types unconstrained since the functional model should be available as soon as possible and is furthermore applied to determine the realisation parameters used to constrain the data types. This is summed up in the following guideline:

```
...
TYPE t_AtomType IS ( INT, SYM );
SUBTYPE t_AtomValue is integer;

TYPE t_Atom IS RECORD
   AtomType : t_AtomType;
   Value : t_AtomValue; -- needs to be
                        -- constrained
END RECORD;
...
```

[2]Many of the principles proposed in this paper are valid for other HDLs, like verilog, as well.

Guideline 3 (Unnecessary Constraints) *Do not constrain values unnecessarily at the functional modeling level.*

The introduction of constraints will be eased, if the type e.g. *t_AtomValue* has been constrained with a constant, which is set to *integer'high*. The code fragment shown above is a compromise between the needs of "rapid-prototyping" to understand the specification and requirements of the design process after the specification has been fixed.

However experience has shown that a central catalogue with all realisation parameters can be useful throughout the whole design process. Such a catalogue should contain for each independent realisation parameter to be determined by simulation: Its **Symbolic name**, its **Value** and the **Data sets** used in the simulations leading to the determination of the value. Whereas for each dependent realisation parameter the catalogue should list the **Symbolic name** and the **Dependency** from the independent realisation parameters.

Such a dependency is given by simple equations as are following for the atom value used above. If the integer part of an atom complies with $integerValue \in [MinAtomI, MaxAtomI]$ and the hash index satifies $hashIndex \in [0, MaxHash]$, then the range of an atom value is given by $value \in [min(-MinAtomI, 0), max(MaxAtomI, MaxHash)]$

Guideline 4 (Central Catalogue) *Establish as soon as possible a catalogue for the whole project with all realisation parameters and maintain it.*

Goal of the design project leading to these guidelines was an add-on board and thus a software environment running on the host computer had to be developed. This software environment comprises a compiler with a tailored back-end and a runtime system [2]. It has been implemented with 27.000 lines of C code.

$MaxHash$ is a parameter of this software environment. The most strange errors have occured after merely a parameter of the software environment had been changed and henceforth the model of the hardware no longer worked correctly. Therefore an additional guideline seems to be useful:

Guideline 5 (Global "stddef" file) *Create a global standard definition file (stddef) accessible by all hardware and software components.*

Each object belongs to an *ObjectLengthClass* determining the number of atoms attached to it. An atom attached to an object is called an attribute of that object. The functional behaviour of this memory can be described by a package containing a procedure to create an object, another to remove an object and a function to retrieve a single attribute value of a given object. In order to ease the access to a single attribute value an object identifier *ObjID* has been introduced. The data structures and the accessing procedures were bundled into a package[3] as shown below[4]:

```
package OM is
   ...
   type t_ObjectMemory, ..., t_Attlist  is array (natural range <>) of t_Atom;
     -- index ranges for the object memory and the attribute list need upper bounds
   subtype t_ObjID, ..., t_AttrIndex is natural;
   subtype t_ObjLengthClass is positive;
     -- The upper bounds for the indices used above are to be used to constrain these types as well
   ...
   procedure OM_CreateObject
     (ObjLengthClass : t_ObjLengthClass; Attributes : t_Attlist;
      ObjID : out t_ObjID; Memory : inout t_ObjectMemory);

   procedure OM_RemoveObject (ObjID : t_ObjID; Memory : inout t_ObjectMemory);

   function OM_GetAtom (ObjID : t_ObjID; attr_index : t_AttrIndex; Memory : t_ObjectMemory) return t_atom;
```

[3] This is an attempt to encapsulate data structures and accessing procedures to construct an *abstract data type* in the sense of object oriented programming [11] with the facilities of VHDL.

[4] The habit to begin any function name with an abbreviation of the package name stems from C. Name conflicts of functions from different packages may be handled in VHDL by overloading ($2.3) or by application of selected names ($6.3) [12].

```
...
end OM;
```

The first idea to construct the data type of the object memory was to use a linked list, which can be implemented in VHDL with *access types* [12]. This would have been the most general implementation and does not force the designer to introduce realisation parameters unnecessarily. Furthermore the creation and removal of objects is easily implemented by a linked list, but the dereference of a single attribute value can be implemented more concisely by an array, thus the declarations as shown above have been used.

In the declaration of the package *OM*, data types have been declared for types which are equally subtypes of integer, as shown in the adjacent code fragment. Furthermore at the functional modeling stage data of type *t_ObjLengthClass* determines the *ObjLengthClass* an objects belongs to and thereby the maximal number of attributes. Whereas data of type *t_AttrIndex* denotes the number of an attribute to be read. Therefore the same data type definition could have been used, but at this stage it is not quite clear, if at a later stage additional attributes have to be added to ease the implementation of hardware for the management of the *OM*.

```
...
package om is
   subtype t_ObjID, ..., t_AttrIndex is natural;
   subtype t_ObjLengthClass is positive;
   ...
end om;
```

The declaration of specific data types for each signal or variable has the advantage that a certain awareness about the need of constraining is created and that later in the design process the introduction of constraints is eased.

Guideline 6 (Specific Data Types) *Declare specific data types and subtypes for all signals, variables and constants.*

3.2 Separation and Migration of the Functionality

The functional model was applied to study the behaviour of the applied algorithms, like the memory allocation algorithm of the case study, and the determination of the various realisation parameters, like the maximum value of an integer in an atom. Concurrently to these simulations, the functionality already described in the functional model as a set of procedures working upon appropriate data structures was separated and migrated to the behavioural models of the first structural model. [3]

The purpose of this separation and migration is the definition of entities, which can be developed by several designers independently. The development of each entity is significantly eased by the fact, that the model of it can be simulated within the whole system environment [2]. Separation and migration proceeds in the following steps:

- **Identification** of the data structures and procedures defining the behaviour of an entity.
- **Command definition** for the various modes to access the data structures. In the case study objects are created and removed via a single port as shown in section 3.2.3. Therefore commands are defined in the following fashion: `type t_OM_command is (CREATE, REMOVE);`
- **Entity declaration** of the new entity using in the *port list* [12] data types mostly available in the definition of the procedures. The reuse of the data types is demonstrated by the following code fragment showing the declaration of the signal *object_data* shown in the figure in section 3.2.3.

```
...
type t_object_data is
  record
    command         : t_OM_Command;     -- create/remove
    ObjLengthClass  : t_ObjLengthClass; -- create
    Attributes      : t_Attlist;        -- create
  end record;
...
```

- **Definition of a protocol** to transfer the input, output data and the commands.

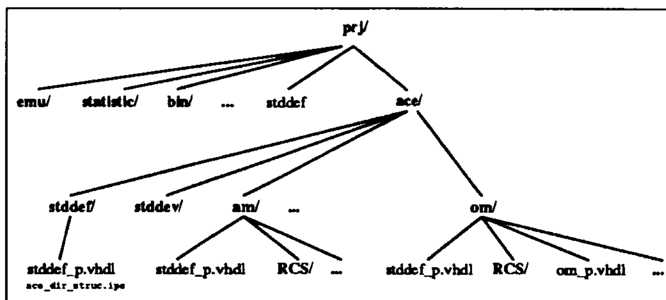
- **Replacing the procedure calls** with sequences as shown in the following code fragment:

```
...
object_data.command <= CREATE;
object_data.ObjLengthClass <= obj.ObjLengthClass;     object_data.Attributes <= obj.Attributes;
object_dav <= '1';
WAIT UNTIL object_dac = '1';
result := ObjID;     object_dav <= '0';
WAIT UNTIL object_dac = '0';
...
```

- **Calling the procedures** Within the architecture body of the new entity a symmetrical sequence is used to handle the protocol and call the procedures appropriately.

3.2.1 Directory Structure

For each component a directory has been created within the file system of the operating system. The overall directory structure is shown in the figure below. The entire directory structure has been hooked at the *prj* directory. The *emu* directory contains the functional model implemented in C, called AcE-Emulator, the *statistic* directory comprises the tools to extract several statistical properties of the system and *bin* the executables particular to the project. *stddef* is the file with the realisation parameters used by the software environment [2] and by the models of the hardware in VHDL. For each hardware component a directory has been create within the directory *ace*. A local standard definition file (*stddef_p.vhdl*) and a VHDL library [12] has been created for each hardware component as well. At the right hand side of the adjacent figure the directory for the component used as the case study is shown. Additionally to the directories for each component a directory containing the global hardware standard definition file (*stddef*) and a directory with some packages of globally relevant entity declarations and functions (*stddev*) has been created.



Each directory with source files has an *RCS* directory to ease the management of the revisions of the source files [13]. From the viewpoint of current software engineering practises, it is an astonishing fact that hardware designers use very seldomly a systematic revision management. This is probably a consequence of the bottom-up design methodology applied by them, which does not make revisions necessary, since a component is designed and afterwards only instantiated. Since a change of a schematic tends to be as costly as a redesign, components are very rarely reused. The implications of generic design to reusability are discussed in section 5.

3.2.2 Local and Global Standard Definition File

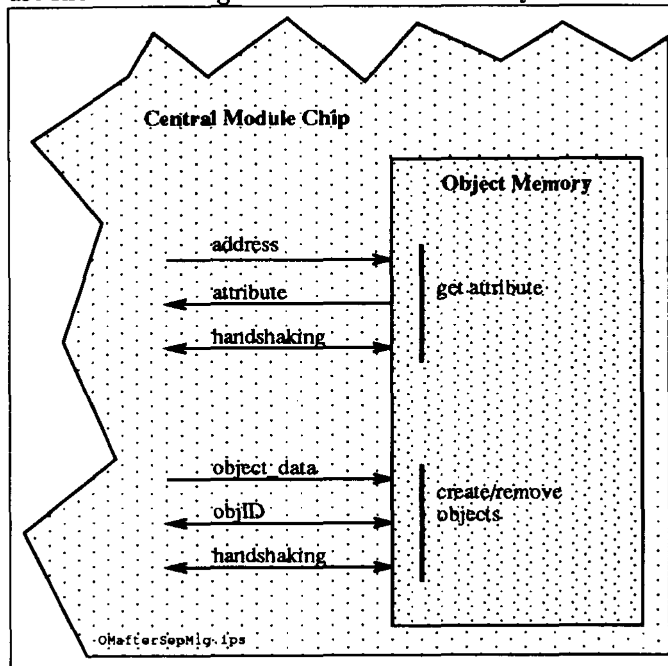
Each directory for a component contains a local standard definition file to limit the size of the global *stddef* file and to reduce the amount of recompilation, if a single realisation parameter has been changed. The decision to declare a data type or constant in the global or local standard definition file depends on the following: A change to a value or an equation in the global *stddef* propagates automatically to all concerned entities, since the global stddef is within the scope of all models. But this propagation is achieved by a costly recompilation of all VHDL models.

Guideline 7 (Local "stddef" File) *A realisation parameter only used by a single entity should be declared in the local standard definition file.*

Most data types of the code fragments in section 3.1.1 are declared in the global standard definition file, since they are used by more than one entity. Only the data type *t_ObjectMemory* is referenced exclusively by the models and packages of the OM, and therefore declared in the local standard definition file shown in the figure in section 3.2.1.

3.2.3 Case Study

Separation and migration of the functionality made clear, that objects are created and removed via one interface and attributes are read via another. The interfaces of the first model of the object memory (*OM*) are shown in the figure below. The new entity can be modeled without the need to introduce any new realisation parameter, since the port list can use data types already available or compositions of them. It is worth noting here, that the signals used in the port list, like the *object_data*, would require an interface with a huge number of bits. Therefore this signal must be transfered in several clock cycles on the real chip. Such signals are called *abstract signals* in the sequel, since they do not have a definite relation to the physical signals of the real chip [3]. However these abstract signals are very useful to postpone implementation decisions until the knowledge to make them properly is available. The transition from abstract signals to concrete signals and the consequences of the application of abstract signals to generic design are discussed in section 4.1. The model outlined in the adjacent figure is to a far extend a generic model, though no generic list has been included in the entity declaration. Every realisation parameter is unconstrained, hence the model can be used in all configurations to validate and verify the other entities thoroughly. The realisation parameters, like the maximal number of attributes in an object, are passed to the entity via the data types of the signals in the port list [12]. This is one of the three possibilities to transfer realisation parameters to an entity discussed in section 3.3.1.



This model was used as a simulation model by the designers developing the other entities of the first structural model and as a reference model for the decomposition and further refinement of the entity applied as a case study.

3.3 Decomposition and further Refinement

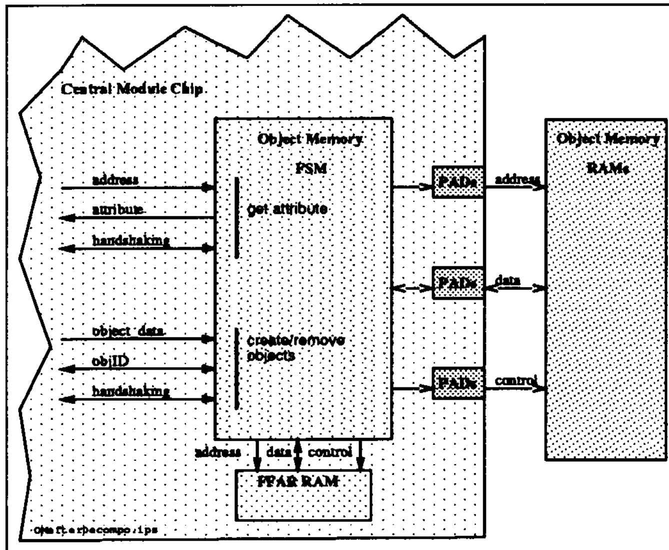
Architectural considerations and simulations with the first structural model have shown that attributes must be dereferenced with a high throughput and a small latency. This can be achieved if all attributes of an object are stored successively in a RAM. The first address of the address range occupied by an object is used as the object identifier (*ObjID*). The address to be supplied to this RAM is then given by: $attributeAddr := ObjID + attributeNumber + offset$. The $offset$ will be necessary to store the house-keeping information attached to each object.

Objects are dynamically created and removed. Considerations of the whole hardware and software system lead to the insight that the creation and removal of objects is neither critical in respect to latency nor throughput, but the memory freed by a removal must be reallocated. Therefore a memory allocation algorithm has been developed reusing the free space created by removal of an object of the same length class.

This allocation algorithm has been implemented in the functional model. Measurements with the functional model sketched in section 3.1.1 indicated, that this memory allocation algorithm has nearly optimal memory allocation properties for the studied benchmarks. This algorithm uses a pointer to the beginning of the free space (*freePtr*) and a linked list containing the free memory blocks of each length class. The *freePtr* pointer and the pointer to the first block in each linked list are stored in registers

called *first free address register* (FFAR).

The realisation parameters of the FFAR, like its width, are an example for a realisation parameter introduced in the course of the design process. The width of the FFAR $FFARwidth$ is given by $FFARwidth \geq \lceil ld(LenObjMem) \rceil$. The number of registers $NoFFAR$ is given by $NoFFAR \geq MaxNoAttributes$. These dependent realisation parameters and their dependencies are defined in the local standard definition file (stddef_p.vhdl) of the directory *OM* shown in the figure in section 3.2.1. Simulations with the functional model determining the realisation parameter *LenObjMem* and evaluation of the available manufacturing technology (1.5 μm ES^2 [14]) have shown that a RAM for the object memory would have fitted on the *central module chip*, where the entity of the case study is located. Furthermore, analysis of the available benchmarks has shown that the number of FlipFlops ($FFARwidth \cdot NoFFAR$) necessary for the FFAR suggest an implementation by an on-chip RAM instead of single FlipFlops. Therefore the entity *Object Memory* shown in the figure in section 3.2.3 has been decomposed into an entity *Object Memory FSM*, an on-chip *FFAR RAM* and several off-chip *Object Memory RAMs* as shown in the adjacent figure.



3.3.1 Transfer of the parameter to each entity

Realisation parameters defined either in the global or the local standard definition file are to be passed to each entity of concern. This can be achieved by passing them to the entity through:

- Its **generic list** [12],
- **constant definitions** in package within the scope of the model of the entity or
- **data types** of the signals in the *port list* [12].

Generic List: In order to illustrate the transfer via a generic list, the entity declaration and instantiation of a RAM model from the global standard device file (*stddev*) is shown below:

```
package stddev is – global standard device file
entity IntegerRAM is
...
   generic(width, length : positive);
   port(clk        : in bit;
        addr       : in natural range 0 to length-1;
        read       : in bit;
        dataIn     : in natural range 0 to 2**width-1;
        dataout    : out natural range 0 to 2**width-1);
end IntegerRAM;
...
end stddev;
```

The entity declared above is instantiated within the architecture body of the OM:

```
library stddev;    use stddev.all;
library math;      use math.all;
library stddef;    use stddef.all;
architecture structure of OM is
...
begin
   FFAR_RAM : IntegerRAM
      generic map(ceil(ld(LenObjMem)),
                  MaxNoAttributes)
      port map (clk, FFARaddr, ..., FFARout);
...
```

The realisation parameters *LenObjMem* and *MaxAttributes* are passed to the entity *OM* through its generic list. These parameter are declared in the global standard definition file. The library *math*

is just a set of function stubs [15] to the floating point library of the operating system. This is the safest approach, since all relevant realisation parameters are visible to the designer instantiating the *IntegerRAM*. However entities with a large structural complexity tend to have a very long generic list, which is hard to edit. Furthermore a new generic, introduced during the decomposition process, must be passed through all necessary levels of hierarchy in the structural model. Thus all entity declarations, all component declarations and all component instantiation statements [12] have to be edited. Experience has shown, that this can be a very tedious and errorprone task.

Constant Definitions: A slightly changed RAM declaration compared to the code fragment above is used to demonstrate the transfer of realisation parameters through constant definitions in a package within the scope of the component declaration. The following code fragment belongs to the global *stddev* file:

```
package stddev is -- global standard device file
entity IntegerRAM is
  port(clk     : in  bit;  ...,  dataout : out natural);
end IntegerRAM;
...
end stddev;
```

Merely the declaration of the generic port has been removed. The following lines of code are from the local standard definition file:

```
library stddef;    use stddef.all;        library math;      use math.all;
package stddef is -- from the local stddef file
  constant FFARwidth  : positive := ceil(ld(LenObjMem));
  constant FFARlength : positive := MaxNoAttributes;
  ...
end stddef;
```

The constants declared in the local *stddef* file are used within the architecture body sketched below to declare a component. This component is bound in a separate configuration specification to the entity declared in the global *stddev*.

```
library OM;  use OM.stddef.all;
architecture structure of OM is
  component FFAR_RAM is
    port(clk : in  bit; addr : in  integer range 0 to FFARlength-1; ...,
         dataout : out integer range 0 to 2**FFARwidth-1);
  end component;
  ...
end structure;
```

This approach avoids the creation and maintenance of long and complex generic lists. Furthermore the dependencies of the realisation parameter are not cluttering the *generic maps* in the structural models, since these dependencies are defined centralised in the local standard definition file. In the case where several instantiations of an entity use the same dependencies, but with altered realisation parameters, different components must be defined in the structural model and the dependencies must be declared multiply in the local *stddef_p.vhdl*. Therefore a risk is created that one of the equations expressing the dependency is not properly maintained during the design process.

Data Types The realisation parameters are passed to the component in the code fragment above through the definition of the data types of the port signals. This method can be extended as shown in the following code fragment of the local *stddef*:

```
library stddef;    use stddef.all;          library math;      use math.all;
package stddef is -- from the local stddef file
  constant FFARwidth  : positive := ceil(ld(LenObjMem));
  subtype t_FFARdata is natural range 0 to 2**FFARwidth-1;
  ...
  constant FFARlength : positive := MaxNoAttributes;
```

```
  subtype t_FFARaddr is natural range 0 to FFARlength-1;
  ...
end stddef;
```

The values of the independent realisation parameters are defined in the global *stddef*, which is included in the scope of the local *stddef*. The independent realisation parameters are used in the local *stddef* file to define the dependent realisation parameters and these are applied to declare data types. These data types are utilized to declare signals as sketched below:

```
use work.stddef.all;      -- local stddef
architecture structure of OM is
  ...
  component Integer_RAM ... end component;
  ...
  signal FFARaddr : t_FFARaddr;    signal FFARout  : t_FFARdata;
  begin
  FFAR_RAM : IntegerRAM  port map (clk, FFARaddr, ..., FFARout);
  ...
end structure;
```

The definitions of realisation parameters and the definition of a structural model are separated with this approach. All dependencies are concentrated in the local standard definition file and the structural model uses the data types defined in the standard definition file to describe the structure of the model in a most concise manner. In the case realisation parameters, like length of queues, can not be passed through data types of signals in the port list, a generic list with e.g. a record of these realisation parameters must be added[5].

The different possibilities to pass realisation parameters to an instantiation are summarized in the following guidelines:

Guideline 8 (Passing via Generic List) *To an entity with a low structural complexity the realisation parameters should be passed via its generic list.*

The transfer via a generic list is recommended for entities with a low structural complexity, since it makes all realisation parameter visible and furthermore this approach is quite simple.

Guideline 9 (Passing via Data Types) *To an entity with a higher structural complexity realisation parameters should be passed as far as possible through the data types of the port signals. The data types of the signals are declared in the local/global standard definition file.*

The main advantage of this approach is the clear separation of the definition of a structural model and the definition of the realisation parameters.

3.3.2 Integration of optimized Models

In the final stage of a design project many realisation parameters have been fixed. Therefore the possibility arises to optimize the models used for logic synthesis for the set of chosen parameters. It is of some importance that the optimized models are integrated with the already available more general ones in such a manner that their generality is not affected. This can be achieved in the following fashion:

Behavioural Models For behavioural models the following code frame should be applied:

```
process(clk, ...)
begin
   if t_addr'high = some_specific_value
   then
      – optimized version
   else
      – generic version
   endif;
end process;
```

Structural Models For structural models a very similar code frame using the *generate statement* can be used to integrate less generic, but optimized versions:

```
architecture structure of FFAR is
   opt_ver : if t_FFARaddr'high < RAM_limit generate
      – register version
   end generate opt_ver;
   gen_ver : if t_FFARaddr'high ≥ RAM_limit generate
      – RAM version
   end generate gen_var;
end structure;
```

[5] Such a record eases the maintenance of the generic lists tremendously, since only the local stddef needs to be edited.

It is worth to be noted, that an attribute is used in the code frame above to extract the value of a realisation parameter from a data type passed to the instantiation via its *port map*. The limitations of the integration of optimized versions are discussed in section 4.2.

Guideline 10 (Integration of optimized Models) *Models optimized for a specific set are to be integrated into the more general models, with* if *and* conditional generate *statements.*

4 Limitations

Generic design is limited by effects introduced by the transition from abstract signals to concrete signals and by the limited possibilities to parameterise in particular structural models.

4.1 Abstract Signals

In section 3.2.3 *abstract signals* are defined as signals without a definite relation to the physical signals of the real chip. The transition from abstract signals to more concrete signals is achieved by the following three steps [3]:

- **Encoding** of leaf components, like enumeration types, and selection of the representation of e.g. integer.
- **Packaging** of leaf components of a composite [12] data type into a bit vector.
- **Slotting** Partitioning of large bit vector into slots, which can be transmitted over the allocated interface.

In order to ease the design of the entities handling the transmission, a leaf component of the abstract data type is not further partitioned during the allocation to a slot. A leaf component is therefore transmitted in a single clock cycle. The necessary length of a slot $MaxLenSlot$ is given by:

$$MaxLenSlot := \max_{\forall slot} \left[\sum_{\forall comp(slot)} lenComp(slot, comp) \right]$$

$lenComp(s, c)$ gives the length of component c in slot s, whereas $comp(s)$ denotes the set of components of slot s. In the case the length of each component $lenComp(slot, comp)$ is varied freely, the tiling effects lead to a waste of silicon area and speed. Therefore the possibilities of a generic design are limited after the transition from abstract to concrete data types has been made.

4.2 Limits of Parameterisation itself

In section 3.3.2 it has been demonstrated how to integrate models optimized for a specific set of realisation parameters into a more general model. Experience has shown that very soon a model begins to become unreadable if too many branches with optimized code are introduced into a model. In order to assess the amount of parameterisation in a model the parameterisation degree PD is defined by:

$$PD := \frac{specific\ Statements}{NoStatements} \cdot 100\%$$

$NoStatements$ denotes the number of statements in the entire model and $specific\ Statements$ gives the number of statements specific to optimized versions of the model. If PD exceeds a specific value it is generally stated, that a new architecture has been developed. In such a case at least a new architecture body should be declared.

5 Reusability

Generic design improves by encapsulation of the design know-how the reusability of a model significantly. Furthermore generic design creates the possibility to extend the point in the design space spread by the realisation parameters of the different architectures to a region around this point. For each model it remains to be seen how far this extension can lead to efficient solutions.

Design is the transformation of an abstract specification into a structural model instantiating available components. Mainly two types of tools to automatically transform a design can be identified:

- **Horizontal tools (wide functional scope, but small transformation step)** used to transform a model for an arbitrary purpose, but at a limited level of abstraction into a less abstract structural model. (e.g. logic synthesis tools)
- **Vertical tools(narrow functional scope, but large transformation step):** are used to transform even a very abstract behavioural model, but with a well defined purpose into a much less abstract model. (e.g. RAM or FIFO generators)

Horizontal tools are beginning to extend their scope from a single clock cycle (RTL level or logic synthesis) to several clock cycles ("high level synthesis"). Vertical tools are currently mainly represented by so-called generators directly creating the layout for memory units or data paths [14,16].

Therefore generic design of a component can be seen as a first step towards the integration of this component into a vertical tool, generating a model which can be efficiently synthesized or "transformed" into a gate level model. By generating a synthesisable model instead of layout such a tool would be technology independent. Furthermore for each entity an abstract generic model used for simulation needs to be implemented.

It has been shown in section 4.2 that parameterisation of a single model is limited, therefore some sort of an *architectural expert system* will be necessary. Such an architectural expert system has to choose the most appropriate architecture and set the realisation parameters accordingly. The combination of a set of generic models for a specific entity with the appropriate *architectural expert system* constitutes an element of a *system design library.*

Our vision of a future system development environment consists of the following parts:

- **Multi level system simulator** to execute and analyse the abstract models written by the system designer [17].
- **System design library** comprising generic models for each relevant entity and an appropriate architectural expert system.
- **Logic synthesis tool** to synthesise among others the chosen and parameterised generic models of the system design library.

That such a vision of a *system design library* is more realistic, than it seems at the first glance can be made clear, if we remind ourselves that a few years ago every designer e.g. working with adders had to know exactly when to use a carry-ripple, carry-look-ahead or carry-skip architecture [18]. Nowadays, the designer using a logic synthesis tool [19] is able to avoid worrying about these different architectures by simply coding e.g. `bitCnt := bitCnt + 1;` into his behavioural model.

Consequently it is expected, that the future system designer no longer needs to be an expert in implementing an FIR filter, a Viterbi decoder or many other system building blocks, since he has access to a *system design library* appropriate for his application domain.

6 Conclusions

The importance of generic design for a top-down design flow has been shown. A set of guidelines, having evolved during a real top-down design project, have been discussed. The implementation of these guidelines has been illustrated with many small code fragments of a case study. Furthermore the limitations of generic design have been enumerated. It has been shown that generic design will increase

the reusability of models. This paper has finished with an assessment of the influence of generic design on future system development environments.

The efficiency of the top-down design flow from specification to sign-off can be increased significantly if the principles of generic design are applied.

7 Acknowledgements

Without the hard work and enthusiasm of the students, who have worked within this design project for their master theses, this project would not have been possible. Furthermore we gratefully acknowledge discussions about the future impacts of generic design with our colleagues: Olaf Joeressen, Peter Zepter and Oliver Mauss.

References

[1] E. McCluskey, *Logic Design Principles.* International Series, Prentice-Hall, 1986.

[2] K. ten Hagen and H. Meyr, "Concurrent design of a chipset and its runtime environment," in *Fifth International ASIC Conference*, pp. 525–528, IEEE, September 1992. ISBN 0-7803-0768-2.

[3] K. ten Hagen and H. Meyr, "Top-Down Design mit VHDL: Ein Erfahrungsbericht," in *GME Fachtagung Mikroelektronik* (D. Seitzer, ed.), pp. 121–126, GME, März 1993. ISBN 3-8007-1934-7.

[4] M. Schoppers, "Real-Time Knowledge Based Control Systems," *Communications of the ACM*, vol. 34, pp. 27–30, August 1991.

[5] J. Wu, T. Young, E. Kawamoto, and W. Keutgens, "Accurate VHDL Libraries for ASIC Design," in *Fifth International ASIC Conference*, pp. 327–330, IEEE, September 1992.

[6] K. ten Hagen, M. Colsman, and C. Schotten, "Tools for Top Down Design," technical report, RWTH-Aachen, IS2 -61 18 10-, Templergraben 55, D-5100 Aachen, Germany, October 1992.

[7] B. Lutter, W. Glunz, and F. Rammig, "Using VHDL for Simulation of SDL Specifications," in *EuroDAC*, pp. 630–635, IEEE, September 1992.

[8] J. Kunkel, "COSSAP: A stream driven simulator," in *IEEE International Workshop on Microelectronics in Communications, Interlaken, Switzerland*, March 1991.

[9] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," *SIGPLAN Notices*, vol. 17, pp. 120–126, June 1982.

[10] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[11] B. Meyer, *Object-oriented Software Construction.* Prentice Hall International, 1988.

[12] IEEE, *VHDL Language Reference Manual*, 1987.

[13] W. F. Tichy, "Design, Implementation and Evaluation of a Revision Control System," in *6th International Conference on Software Engineering*, IEEE, September 1982.

[14] European Silicon Structures, Rousset, France, *ES2 ECPD12 & ECPD15 Library Databook*, 1992.

[15] Vantage Analysis Systems, 42808 Christy Street, Suite 200, Fremont, CA 94538, USA, *User's Guide*, April 1992.

[16] R. Jain, P. T. Yang, and T. Yoshino, "FIRGEN: a computer-aided design system for high performance FIR filter integrated circuits," *IEEE Trans. on Signal Processing*, vol. 39, pp. 1655–1668, July 1991.

[17] P. Zepter and K. ten Hagen, "Using VHDL with stream driven simulators for digital signal processing applications," in *EURO-VHDL'91 Proceedings*, (Stockholm, Sweden), pp. 196–203, September 8-11 1991.

[18] K. Hwang, *Computer Arithmetic.* Wiley, 1979. ISBN 0-471-03496-7.

[19] Synopsys Inc., Mountain View, CA, USA, *Design Compiler Manual.*